

Parallelized Distributed Embedded Control System for 2D
Walking Robot for Studying Rough Terrain Locomotion

By

Gavin Strunk

Submitted to the graduate degree program in Mechanical Engineering and the
Graduate Faculty of the University of Kansas in partial fulfillment of the
requirements for the degree of Master of Science.

Chairperson Dr. Terry Faddis

Dr. Sara Wilson

Professor Robert Umholtz

Date Defended: _____

The Thesis Committee for Gavin Strunk
certifies that this is the approved version of the following thesis:

Parallelized Distributed Embedded Control System for 2D Walking Robot for
Studying Rough Terrain Locomotion

Chairperson Dr. Terry Faddis

Date approved: _____

Abstract

Biped robots present many advantages for exploration over mobile robots. They do not require a continuous path, which allows them to navigate over a much larger range of terrain. Currently, bipeds have been successful at walking on flat surfaces and non-periodic rough terrain such as stairs, but few have shown success on unknown periodic terrain.

The Jaywalker is a 2D walker designed to study locomotion on uneven terrain. It is a fully active robot providing actuation at every joint. A distributed, parallelized, embedded control system was developed to provide the control structure for the Jaywalker. This system was chosen for its ability to execute simultaneous tasks efficiently. The two level control system provides a first level to implement a higher level control strategy, and a second lower level to drive the Jaywalker's systems.

The concept was implemented using the Parallax Propeller chip for its relative fast clock frequencies and parallel computing functionality. The chips communicate over a new variation of the I²C bus, which allows multiple slaves to listen to information simultaneously reducing the number of transmissions for redundant data transfers.

The system has shown success in taking steps with open loop control. The success of the step is highly dependent on the initial step length using open loop control, but this dependency can be eliminated using closed loop control. The robust structure will provide an excellent platform for uneven terrain locomotion research.

Acknowledgments

I would like to extend a special thanks to my graduate advisor, Dr. Terry N. Faddis, for his guidance and support during this research. I would also like to thank my committee members, Dr. Sara E. Wilson and Professor Robert C. Umholtz for their support throughout this research.

Finally, I would like to thank my family and friends whose support has been vital to my success. I would like to give special thanks to my parents, Richard and Benita, for their advice and support. To my sister Sara, your support was also very much appreciated.

Thanks.

Table of Contents

LIST OF FIGURES.....	vi
NOMENCLATURE.....	ix
INTRODUCTION.....	1
1. EMBEDDED CONTROL THEORY FOR BIPED WALKER ON UNEVEN TERRAIN.....	8
1.1 Advantages of Embedded Control.....	8
1.2 Distributed Control.....	9
1.3 Parallel Computing.....	12
1.4 Jaywalker Control Structure.....	14
1.5 Feedback Capabilities.....	17
1.6 Control Stages of 2D Bipedal Walking.....	19
2. IMPLEMENTATION OF THE EMBEDDED CONTROL SYSTEM USING THE JAYWALKER TEST BED.....	23
2.1 Jaywalker Mechanical Configuration.....	23
2.2 Jaywalker Electronic Hardware.....	26
2.3 Circuit Design and PCB Layout.....	30
3. JAYWALKER SOFTWARE.....	31
3.1 Multi-Master Parallel Slave I ² C Bus.....	31
3.2 Emergency BUS.....	37
3.3 Jaywalker GUI.....	38
4. JAYWALKER TESTING.....	41
4.1 Motor Ramping.....	41
4.2 Range of Motion.....	42
4.3 Data Acquisition.....	42
4.4 Step Testing.....	43
CONCLUSIONS AND RECOMMENDATIONS.....	44
APPENDIX A: JAYWALKER CONTROL FLOWCHART.....	46
APPENDIX B: JAYWALKER WIRING SCHEMATICS.....	47
B.1: Level 1 Main Controller Wiring Diagram.....	47
B.2: Level 2 GUI Controller Wiring Diagram.....	48
B.3: Level 2 Power Controller Wiring Diagram.....	51
B.4: Level 2 Foot Controller Wiring Diagram.....	54
B.5: Level 2 Sensor Controller Wiring Diagram.....	56
B.6: Connector Board Wiring Diagram.....	58
B.7: Jaywalker DB-25 and Round Wiring Diagrams.....	62

APPENDIX C: JAYWALKER PCB LAYOUTS.....	67
C.1: Main Controller Layout.....	67
C.2: GUI Controller Layout.....	68
C.3: Power Controller Layout.....	69
C.4: Foot Controller Layout.....	70
C.5: Sensor Controller Layout.....	71
C.6: Connector Board Layout.....	72
APPENDIX D: JAYWALKER PROGRAM CODE.....	73
D.1: MD1 PWM Driver.....	73
D.2: Jaywalker GUI.....	75
D.3: MMPS I2C Driver.....	83
APPENDIX E: JAYWALKER GUI SCREENSHOTS.....	88
E.1: Walk Menu.....	88
E.2: System Check Menu.....	89
E.3: Manual Mode Menu.....	90
E.4: Debug Menu.....	91
APPENDIX F : JAYWALKER STEP SCREENSHOTS.....	92
F.1 Middle Leg Step Screenshots.....	92

List of Figures

Figure 1.2.1: Distributed Control System applied to fan control.....	10
Figure 1.3.1: Problem solving comparison between serial and parallel computing...	12
Figure 1.3.2: Example problem using SIMD configuration.....	13
Figure 1.3.3: Example problem using MISD configuration.....	13
Figure 1.3.4: Example problem using MIMD configuration.....	14
Figure 1.4.1: Block diagram of 2 level control structure for Jaywalker.....	16
Figure 1.4.2: Block Diagram of parallelized 2 level control structure.....	17
Figure 2.1.1: CAD drawing of the Jaywalker Test bed [1].....	23
Figure 2.1.2: CAD drawing of the iHD Test bed [1].....	24
Figure 2.1.3: CAD drawing of the Jaywalker knee [1].....	24
Figure 2.1.4: CAD drawing of HPAA [1].....	26
Figure A.1: Jaywalker Control System Flowchart.....	46
Figure B.1.1: Propeller wiring diagram with EEPROM and connectors corresponding to GUI Controller connectors.....	47
Figure B.1.2: Regulator power source provided +3.3 VDC and +5 VDC for the Main Controller.....	48
Figure B.2.1: Propeller wiring diagram with EEPROM and connectors corresponding to the connector board.....	49
Figure B.2.2: Regulator power source provided +3.3 VDC and +5 VDC for the GUI Controller.....	50
Figure B.3.1: Propeller wiring diagram with EEPROM, optoisolator circuit for direction pins on MD1 motor drivers, and connectors corresponding the connecting board.....	51
Figure B.3.2: Regulator power source provided +3.3 VDC and +5 VDC for the Power Controller.....	52
Figure B.3.3: Optoisolated transistor circuit to drive the +24 VDC solenoids for the air cylinders.....	53
Figure B.4.1: Propeller wiring diagram with EEPROM and connectors corresponding to the Sensor Controller connectors.....	54
Figure B.4.2: Regulator power source provided +3.3 VDC and +5 VDC for the Foot Controller.....	55
Figure B.5.1: Propeller wiring diagram with EEPROM and connectors corresponding to the connector board.....	56
Figure B.5.2: Regulator power source provided +3.3 VDC and +5 VDC for the Sensor Controller.....	57
Figure B.6.1: Wiring diagram for the connector board including slots for a VGA adapter, SD card adapter, and receptors for the GUI, Sensor, and Power Controllers.....	58
Figure B.6.2: Regulated power source providing +3.3 VDC, +5 VDC, and +24 VDC	59
Figure B.6.3: Connector for PropPlug to program each of the five controllers.....	60
Figure B.6.4: 50 pin edge connectors wiring diagram.....	61
Figure B.7.1: DB25 connector in slot 1.....	62
Figure B.7.2: DB25 connector in slot 2.....	63

Figure B.7.3: DB25 connector in slot 3.....	64
Figure B.7.4: DB25 connector in slot 4.....	65
Figure B.7.5: Round connectors in slots 8 and 9.....	66
Figure C.1: Main Controller PCB layout in ExpressPCB.....	67
Figure C.2: GUI Controller PCB layout in ExpressPCB.....	68
Figure C.3: Power Controller PCB layout in ExpressPCB.....	69
Figure C.4: Foot Controller PCB layout in ExpressPCB.....	70
Figure C.5: Sensor Controller PCB layout in ExpressPCB.....	71
Figure C.6: Connecting Board PCB layout in ExpressPCB.....	72
Figure E.1: Screenshot of the Walk Menu in the Jaywalker GUI.....	88
Figure E.2: Screenshot of the System Check Menu in the Jaywalker GUI.....	89
Figure E.3: Screenshot of the Manual Mode Menu in the Jaywalker GUI.....	90
Figure E.4: Screenshot of the Debug Menu in the Jaywalker GUI.....	91
Figure F.1.1-6: Screenshots 1-6 of Middle Leg Step.....	92
Figure F.1.7-13: Screenshots 7-13 of Middle Leg Step.....	93

Nomenclature

ACK	Acknowledge
ADC	Analog/Digital Converter
AIT	Acknowledge In Turn
ANN	Artificial Neural Network
CAD	Computer Aided Design
CAN	Controller Area Network
CNS	Central Nervous System
CPU	Central Processing Unit
DB-25	D-Subminiature 25 Pin
DIP	Dual Inline Package
DSP	Digital Signal Processor
EBUS	Emergency Bus
EEPROM	Electrically Erasable Programmable Read Only Memory
FPGA	Field Programmable Gate Array
GPIO	General Purpose Input/Output
GUI	Graphical User Interface
HPA	High Priority ACK
HPAA	Hybrid Parallel Ankle Actuator
I ² C	Inter-Integrated Circuit
iHD	Independent Hip Drive
IPM	Inverted Pendulum Model
KB	KiloByte
LCD	Liquid Crystal Display
LEGS	Leg Extension Guidance System
MEMS	Microelectromechanical Systems
MHz	Mega Hertz
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MMPS	Multi-Master Parallel Slave
MSB	Most Significant Bit
PCB	Printed Circuit Board
PD	Proportional Derivative
PWM	Pulse Width Modulation
RAM	Random Access Memory
ROM	Read Only Memory
SCL	Serial Clock Line
SD	Secure Digital
SDA	Serial Data Line
SIMD	Single Instruction Multiple Data
SPI	Serial Peripheral Interface
VDC	Voltage Direct Current
VGA	Video Graphics Array
VM	Virtual Model
ZMP	Zero Moment Point

Introduction

Bipedal walking robots present many advantages over wheeled robots for exploration. Biped robots are designed to traverse terrain using footholds rather than a continuous path, which is synonymous with human locomotion [1]. This method is advantageous because often a continuous path is unobtainable or requires significantly longer traveling distances to reach the same destination. Terrain such as stairs exemplifies an environment conducive to using discrete footholds for locomotion [2]. Significant research has been successfully conducted in navigating flat surfaces and periodic terrain, such as stairs, as seen with robots such as Honda's Asimo [3]. Walking robots have also demonstrated the ability to carry heavier loads than can be done by humans [4]. In addition to assisting humans, advances in robotic walking also aids in more robust prosthetic limb development.

Despite these advances in walking technology, current walking robots do not have the ability to compensate for variation in terrain that is unpredictable. This is an essential ability for robots with aims of being integrated into human environments. The hole in walking logic comes from the limitation of robotic vision systems. Vision systems are increasingly efficient at detecting objects and planning avoidance paths, but this presents several limitations when the robot cannot see an object. It is not difficult for a robot to navigate around a large object like a box or chair as these are easily detected, but what happens with small objects like a toy car or an electrical cord. In many cases these small objects can be detected, but require the robot to

spend a significant portion of time watching the ground, limiting the use of the vision system for other perceptive tasks, such as finding a lost object [5-6]. The other scenario is terrain variations that cannot be detected at all by a vision system, such as ground changes under grass. A freshly cut lawn would appear to be level to a vision system, but many people have had the phenomena of stepping in a hole in a lawn that cannot be seen. These situations are the focus of the research performed at the University of Kansas Intelligent Systems and Automation Lab. This research aims to present a novel embedded control structure that will allow the study of falling on uneven terrain as well as having the robustness to implement corrective control strategies for these scenarios.

A variety of control strategies have been used to control the locomotion of biped walkers. Popular method selections are the Inverted Pendulum Model (IPM), Zero Moment Point (ZMP), Virtual Model (VM), and a variety of artificial intelligence algorithms. Though each has shown some success each has a variety of drawbacks for implementation. IPM aims to represent the robot as a series of lumped masses and calculate trajectories of the joint movement using dynamics equations [7]. This method can be very efficient and accurate if the model is simple and represents the true system well. Unfortunately, many times the models used are not accurate at representing the true system dynamics and accurate models result in very complex equations that cannot be solved efficiently. It is also difficult to conform to the assumptions made to use this model such as the relation of mass at the foot to the mass at the hip. These assumptions are difficult to conform to because of the large amounts of torque required at the feet to walk. When these assumptions are broken

the performance of the controller suffers severely [7]. A variety of other methods aim to control the trajectory of the joints or lumped mass with higher order equations such as polynomials or splines, but each suffers in a similar way [8-9]. These methods provide a fast first approximation, but are generally not realistic to implement in a real time situation due the computational load. To combat this drawback, research has been done to calculate the trajectories offline and perform error tracking online to control the walking robot [9]. This method has shown excellent results when the terrain is known and relatively little variation occur, however it lacks the ability to recalculate the trajectory quickly and therefore will fail on unknown terrain.

One of the most popular control methods is known as the ZMP method. The high level concept is to control the moments at each joint and ensure they sum to zero thereby ensuring stability at every point in time at every joint. This means that at any given point during the walking cycle the robot could be stopped and would be stable, or statically stable [10]. Although this brute force method of control has proven the ability to walk, navigate stairs, and even run with the Asimo robot, it lacks the flexibility to adjust to unknown terrain efficiently because it requires intensive computational time to recalculate joint angles. It is also known that this method is energy intensive compared to human gait and limit cycle walkers that do not require static stability. It has been proven that static stability is not a requirement of stable walking as long as dynamic stability is achieved [11]. Research has also been conducted to reduce trunk movement by adding Fuzzy Logic to the ZMP method, and despite some success the control strategy was not able to relax the strict conditions of the ZMP model [12].

A completely different approach is to use a representation of the real components with virtual equivalents known as the Virtual Model (VM). The VM uses mechanical components to imitate the actual dynamics of the system being controlled. For example, a PD controller can be represented by a spring-damper system. The VM has numerous advantages including intuitive model construction and efficiency compared to trajectory models. The VM also has the ability to make use of adaptive and learning elements to provide flexibility in the control structure [13]. The drawback of the VM approach comes from the fundamental principle of replacing the system with virtual components. VMs can fight the natural dynamics of the system in extreme situations, which is very inefficient and can cause instability. The VM also has a tendency to overcompensate which can cause saturation and instability. Finally, the VM is dependent on the system's ability to track the trajectories that are calculated by the model [13]. Virtual modeling has seen excellent results, but requires a significant development effort as modeling a complex system with equivalent virtual components is not a trivial task.

Fuzzy logic has also been an area of interest for robotic walker controllers. Fuzzy logic aims to relax the boundary definitions as compared to discrete logic. The difficulty with implementing fuzzy controllers is the correct membership function selection and implementation. Typically this process requires the designer to have an in depth understanding of the system previously to accurately select the membership function and verify the results. Fuzzy logic alone has many advantages, but is limited in that it is meant to relax operating conditions which may not be enough to capture the entire behavior of the system [14]. This property makes fuzzy logic a more

popular choice to supplement other algorithms to improve computational speed and increase accuracy.

Artificial intelligence aims to capture the dynamic behavior of the system in a more efficient manner than solving the complex differential equations that represent the system. There are numerous algorithms that have been attempted, but many have not progressed beyond simulations. Artificial intelligence has also shown success when the computational effort is controlled and the convergence of the algorithm is accurate and guaranteed. One of the most popular control methods is Artificial Neural Networks (ANN). ANNs have been implemented in a variety of forms including the addition of fuzzy logic or genetic algorithms to boost the performance of the controller. The main strategy when implementing an ANN is to view the control variable or variables as undergoing a nonlinear transformation. The output then becomes a nonlinear summation of the input variables [15]. The performance of the ANN is highly dependent on the training of the system. It is an iterative process to successfully capture the behavior of the system without memorizing the data set.

Researchers have also explored adaptive methods of biped control. As with many learning algorithms, adaptive algorithms suffer from being computationally intensive and some require a substantial number iterations to successfully converge. In attempt to combine the two methods, researchers have attempted to integrate stochastic processes with ANNs to implement an adaptive learning algorithm [16]. Despite simulation success of these methods, many have not been able to scale their algorithms to a test bed or a viable biped robot.

The scope of this thesis encompasses the development of an embedded control system implemented on the Jaywalker test bed. The first goal of this research is to develop an embedded control structure that is robust enough to improve the walking performance as well as improve data collection rates. Secondly, the control is designed to have a structure flexible enough to support a variety of algorithms to correspond with the stages of research to be performed. Chapter 1 describes the advantages of an embedded controller in achieving the research aims as well as the theoretical control theory of the system. Chapter 2 gives a brief overview of the construction of the Jaywalker test bed as it applies to the control strategy, the electronic component selection and setup, and the software used to interface with the various hardware components and sensors. Finally, Chapter 3 outlines the results and conclusion of the setup as well as future recommendations and improvements to be made.

References

- [1] Vaughan, C. L. (2003). "Theories of bipedal walking: an odyssey." *Journal of Biomechanics* 36(4): 513-523.
- [2] Daley, M. A. (2008). "Biomechanics: Running Over Uneven Terrain Is a No-Brainer." *Current Biology* 18(22): R1064-R1066.
- [3] http://world.honda.com/ASIMO/technology/walking_02.html. Date Accessed: 23 March 2010.
- [4] Choi, H.-S., You, S.-S., Lee, S.-J., Kim, B.-G., Lim, G.-W., Ko, D.-Y. and Moon, W.-J., 2006, "A study on a new biped robot supporting heavy weight," Busan, South Korea, pp. 1180-1184.
- [5] Krell, J. and A. E. Patla (2002). "The influence of multiple obstacles in the travel path on avoidance strategy." *Gait & Posture* **16**(1): 15-19.
- [6] Erin, B., R. Abiyev, et al. (2010). "Teaching robot navigation in the presence of obstacles using a computer simulation program." *Procedia - Social and Behavioral Sciences* **2**(2): 565-571.

- [7] Albert, A. and Gerth, W., 2003, "Analytic path planning algorithms for bipedal robots without a trunk," *Journal of Intelligent and Robotic Systems: Theory and Applications*, **36**(2), pp. 109-127.
- [8] Cisneros, M., Cuevas, E., and Zaldivar D. 2008. "Walking Control Algorithm based on Polynomial Trajectory Generation." CUCEI.
- [9] Hurmuzlu, Y., F. Génot, et al. (2004). "Modeling, stability and control of biped robots--a general framework." *Automatica* 40(10): 1647-1664.
- [10] Ito, S., Amano, S., Sasaki, M. and Kulvanit, P., 2008, "A ZMP feedback control for biped balance and its application to in-place lateral stepping motion," *Journal of Computers*, **3**(Compendex), pp. 23-31.
- [11] Alexander, R. M. (2007). "Biomechanics: Stable Running." *Current Biology* 17(7): R253-R255.
- [12] Park, J. H. (2003). "Fuzzy-logic zero-moment-point trajectory generation for reduced trunk motions of biped robots." *Fuzzy Sets and Systems* **134**(1): 189-203.
- [13] Chae H. An, Christopher G. Atkeson, and John M. Hollerbach. *Model-Based Control of a Robot Manipulator*. MIT Press, Cambridge, MA, 1988.
- [14] Bebek, B. and Erbatur, K., 2003, "A Fuzzy System for Gait Adaptation of Biped Walking Robots," in *Proceedings of IEEE Conference on Control Applications (CCA)*, Istanbul, Turkey, **1**, pp. 669-673.
- [15] Duysens, J., H. W. A. A. Van de Crommert, et al. (2002). "A walking robot called human: lessons to be learned from neural control of locomotion." *Journal of Biomechanics* 35(4): 447-453.
- [16] Huang, H., D. W. C. Ho, et al. (2007). "Robust stability of stochastic delayed additive neural networks with Markovian switching." *Neural Networks* 20(7): 799-809.

1. Embedded Control Theory for Biped Walker on Uneven Terrain

1.1 Advantages of Embedded Control

There are a variety of control hardware selections when developing a control system. Each structure has its own advantages that can be applied successfully to different applications. For example, PCs are excellent for quick development time and ease of use. PCs lack the ability to be energy efficient due the large amount of overhead hardware and software they support, and they are not meant to be packaged onboard a robot typically. Embedded systems are excellent at providing flexibility and can be designed into much smaller packages [1]. They do suffer from significantly lower clock speed compared to most PC setups limiting the computational power.

Embedded systems are a large category of devices, but for the purposes of the Jaywalker robot a microcontroller setup is the topic of discussion. Other devices such as Digital Signal Processors (DSP) or Field Programmable Gate Arrays (FPGA) are also embedded devices, but have a significantly higher cost than most microcontrollers. Microcontrollers in general also have a much lower price tag over a PC based control system [1]. The savings, however; comes at the expense of the designer's time. Typically embedded systems have limited built in functionality to begin with. Luckily with the popularity of object oriented style development there are resources available to avoid rewriting common low level code, such as many communication types, saving the designer large amounts of time.

Another factor when selecting a system for mobile robotics is energy consumption. Biped robots in the future will have to carry their power source in some form, which increases the need to have control systems that can operate on lower amounts of energy to extend the operating time of the robot [2]. The reduced overhead of a large number of peripherals and the lack of an operating system allows microcontrollers to operate on low amounts of energy. Overall, embedded control systems are ideal as the advantages outweigh the disadvantages, but they have not been commonly used because of their relatively low computational capacity compared to a PC.

1.2 Distributed Control

Accounting for the slower processor speeds is essential to an embedded system being successful, especially in a complex application that requires a large number of calculations or includes a large amount of hardware like a robot. This can be accomplished by spreading out the computational load using a distributed control strategy. The distributed control system allows the overall control system to be broken down into smaller pieces that do not require as significant of a computation load [3]. These pieces can be dedicated to a processor increasing the overall performance by requiring less of each processor. This concept has been successfully implemented in a variety of industries, such as manufacturing and industrial processes [4-5]. Figure 1.2.1 shows an example of a distributed system setup applied to fan control:

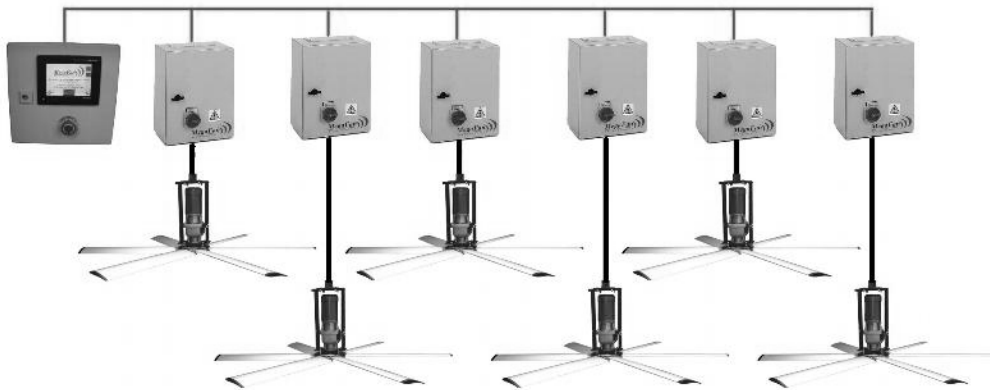


Figure 1.2.1: Distributed Control System applied to fan control.
<http://www.megafans.co.uk/16.html>

The implementation of a distributed control system can become complex as a processor acts as an individual controller, but is also typically part of a larger control scheme. This requires a large portion of custom development around each individual controller because each application can be different leaving very limited options for purchasing off the shelf controllers. An advantage of custom controllers, however; is the ability to optimize their performance to the specific application rather than focusing on generality of operation. The optimization is very advantageous to systems requiring stringent control requirements [5]. Once the individual controllers have been implemented that still leaves out an important piece of every distributed control system.

Normally it is not acceptable to have each controller run independently of each other controller. This can essentially turn a process as a whole into open loop control system even if each controller itself is closed loop. Rarely is open loop control a desirable strategy, but this is especially true of distributed systems. Because

the controller can essentially operate independently of one another it is imperative there is a means of communications. Without a communication bus there is no reference for timing events relative to other events, which is an essential function in many applications [6]. There are an enormous number of options when implementing a communication bus between controllers. Typically, common industries use common communication protocols, such as the CAN bus for the automotive industry and Ethernet for many industrial processes, but this is not a requirement. Several factors are considered when selecting a specific protocol: speed or throughput rate, distance the signal must travel, peripherals on the bus, and even number of wires used [6]. There is not a single correct solution for an application, so selecting a communication bus is simply done by matching the needs of the individual system to a corresponding protocol that meets the needs.

The final component in a distributed system is the main controller. It is the interface between the communication bus and the individual controllers. There are a variety of implementations of a main controller ranging from as simple as a communication host to containing an overarching control algorithm. The complexity of the main controller is highly dependent on the application. It is common to use the main controller as the communication host and as a user interface for the operators of the system [6]. Despite the computational advantages provided by a distributed control system, sometimes this structure is still unable to overcome the at least order of magnitude slower computational speed of the microcontrollers versus a PC. To provide further efficiency parallel computing can be implemented.

1.3 Parallel Computing

Parallel computing in its most simplistic definition is the simultaneous use of computational resources to solve a problem [7]. Figures 1.3.1 shows a graphical representation of the conceptual difference between serial and parallel computing.

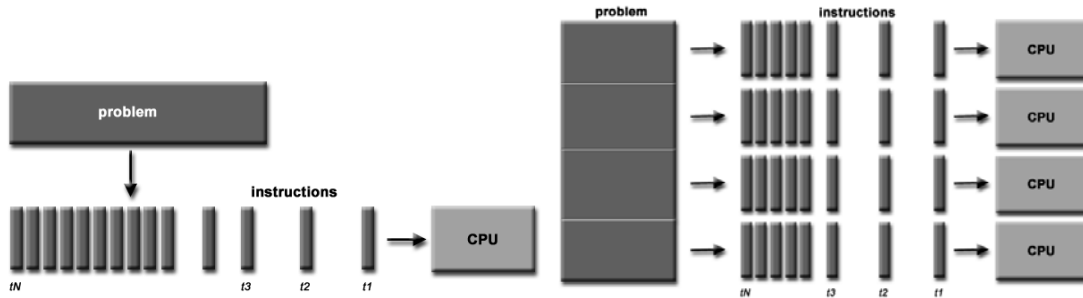


Figure 1.3.1: Problem solving comparison between serial and parallel computing.
https://computing.llnl.gov/tutorials/parallel_comp/

There are a variety of parallel computers that are generally lumped into three categories known as Flynn's Taxonomy: Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD), and Multiple Instruction Multiple Data (MIMD) [7]. SIMD systems execute the same instruction on any given clock cycle, but do so on separate data elements. These systems are advantageous if a common computation needs to be executed on a large data set, such as graphics or image processing problems with high dimensionality [7]. Figure 1.3.2 shows an example of executing the same multiplication instruction on different elements in the same data structure.

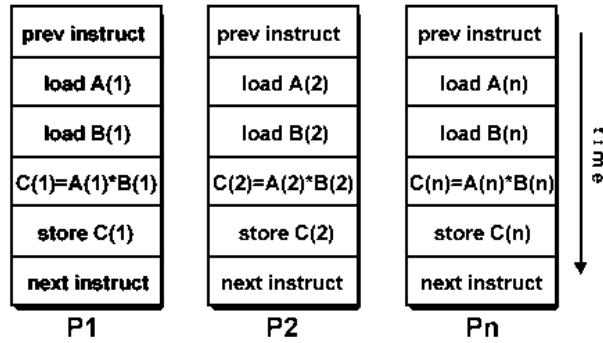


Figure 1.3.2: Example problem using SIMD configuration
https://computing.llnl.gov/tutorials/parallel_comp/

MISD uses a single data stream that performs independent instructions on a common data stream. This is the least common configuration, but could be well suited in brute force applications like cryptography that are running various algorithms on the same data set simultaneously [8]. Figure 1.3.3 demonstrates the ability to execute different functions on the same number in a data set.

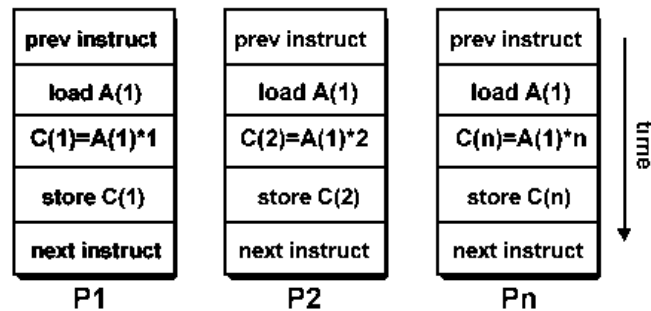


Figure 1.3.3: Example problem using MISD configuration
https://computing.llnl.gov/tutorials/parallel_comp/

The last and most common configuration is MIMD. MIMD allows individual instructions to be performed on independent data sets simultaneously. This provides the most flexibility allowing simultaneous processes to be performed that do not need to be related to each other. The MIMD structure can run synchronously or asynchronously, as well as the ability to operate deterministically or non-deterministically [8]. MIMD is unique in that its structure fits a much broader range

of application due to its flexibility. Figure 1.3.4 shows an example of operating independent functions on independent elements in the data set.

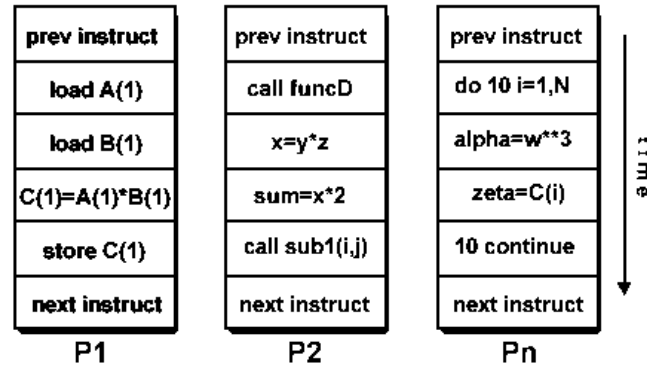


Figure 1.3.4: Example problem using MIMD configuration
https://computing.llnl.gov/tutorials/parallel_comp/

Fortunately, MIMD systems are fairly common and have an ideal setup for biped robotic control as many tasks need to run simultaneously. Using parallel computing strategies to run tasks increases the effective speed of the system compensating for the underpowered components that typically make up the system [9]. The next step is to combine the concept of the distributed control system and parallel computing to the biped robot application.

1.4 Jaywalker Control Structure

The Jaywalker embedded control system is comprised of two levels of control similar to the distributed control system described previously. The first level of control serves as the main controller and holds the high level control scheme implemented for the robot. Then connected by a communication bus, the second level of control drives the low level functionality of the hardware implemented on the robot. This distributed structure allows the system to adapt as the research adapts requiring only the first level of control to change when implementing a new control

algorithm. This allows a robust design over the hardware controllers, while maintaining flexibility at the highest level reducing the development time for new algorithms.

This scheme does require some planning on the designer's part in the future because the high level control strategy is intended to drive the second level of controllers rather than directly affecting the low level actuation. For example, to implement a basic feedback control using an encoder would use the sensor reading as an input and could adjust the position of the robot or the velocity of its movement. This logic would then be communicated to the second level that would execute the necessary motor movements and actuation to achieve the results. This allows the hardware and software of the system to stay modular throughout the research stages.

The second level controllers have been broken up by the following functions: Power controller, Sensor controller, GUI controller, and Foot controller. The power controller handles all of the power systems including stepper motors and air actuators, as well as, the encoder readings. It directly interfaces with micro-stepping motor drivers to provide a fine tune controller over the motor systems. The sensor controller handles the various data acquisition incorporated into the robot, as well as, streaming the data for storage and post processing analysis. The GUI controller handles the user input and output to run a variety of testing modes that will be explained in more detail in Chapter 2. Finally, the foot controller handles the future development of a smart foot used to detect unstable conditions. Figure 1.4.1 shows

the basic outline of the Jaywalker control configuration and Appendix A gives a detailed system flowchart.

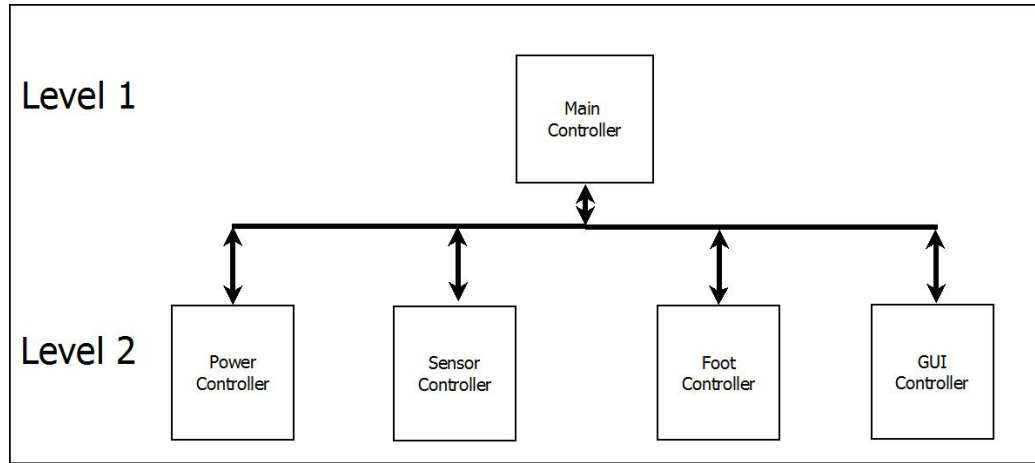


Figure 1.4.1: Block diagram of 2 level control structure for Jaywalker.

The next development step was to parallelize the individual controllers. Each of the level 2 controllers requires simultaneous tasks to be performed. PCs and many microcontrollers use multitasking and threading to accomplish this task, but because the computational requirement of many of the tasks being performed is not intensive at the low level it is faster to dedicate these tasks to their own processor. It was chosen to set up the system in a MIMD configuration because of the lack of redundancy in the data set and the instruction execution. Figure 1.4.2 shows the updated structure where each subdivided block in each block represents an individual processor within the controllers.

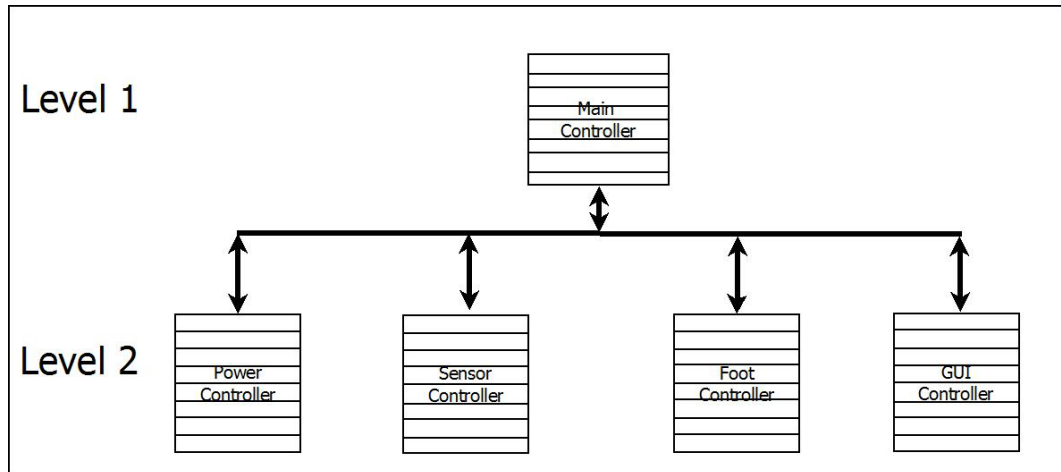


Figure 1.4.2: Block Diagram of parallelized 2 level control structure.

In addition to the structure being robust in handling the numerous tasks for bipedal walking, but it also allows flexibility in the type of the control scheme implemented.

1.5 Feedback Capabilities

Humans are very effective walkers on uneven terrain because we are capable of correcting our instability in two different ways, neural response and reflexes. Neural responses occur when our body assesses our internal sensors and calculates a correction and sends the signals to the appropriate muscles. This is the slower of the two responses and is on the order of a few hundred milliseconds to complete [10]. This type of response is much easier to imitate because microcontrollers can complete a large number of calculations in that amount of time. Unfortunately, for designers, humans typically use neural responses to correct for slight changes and mild instability. The true power of human's locomotion is reflex responses.

Reflexes are responses that do not use neural computation. They are extremely fast typically occurring in 30-45 milliseconds [10]. These responses are

used by our body in response to more extreme conditions, like the doctor's knee tap test. The logic of reflexes is built into the Central Nervous System (CNS), which allows them to respond without neural intervention. Unfortunately, these responses are difficult to imitate because it can take an air cylinder 30 milliseconds to fully charge leaving nearly no time for computing a correction [10]. Despite, this drawback the proposed structure allows for a similar type of control operation as humans.

The equivalent to the neural responses in humans is handled by the main controller or level 1. The main controller will handle responses that have enough time to read the sensor data, make a decision, and relay the correction to the second level. The speed of this operation is heavily dependent on the communication rate between levels of control, which will be discussed in Chapter 2. This provides the first type of feedback for the system, which will be referred to as a level 1 feedback.

The second human response of reflexes will be imitated using what is termed a level 2 feedback. Level 2 feedback will occur within the level 2 controllers directly from their associated peripherals and sensors. These responses will be able to react much quicker than the level 1 response providing a fast response for more extreme instability conditions. Because the level 2 controllers do not contain the overall control scheme their response to the system will be limited to special situations. These special situations are defined by the designer and will come out during fall testing. The idea is to correct the system with the level 2 feedback enough to allow a level 1 feedback to calculate a corrective response and right the system. Many times

when humans begin to fall we do not always correct our gait within a single step. This act of using multiple steps or a sequence of corrective actions will be the aim of the multi-layered feedback approach. This will be an imperative ability of a successful biped because they are inherently not as dexterous as humans.

The key to successful implementation of the level 2 feedbacks will be the ability to characterize unstable behavior rather than attempting to capture all combinations of specific conditions. That approach would be unmanageable and would defeat the goal of providing a fast response to allow the system more time to make corrective actions. Before attempting to implement an algorithm for rough terrain it is important to first navigate a flat known surface to better understand the behavior of the state variables.

1.6 Control Stages of 2D Bipedal Walking

The stages of the gait cycle are very well defined, but the type of robotic walker changes the control strategy. Passive robots require little to no control, but are very limited in their functionality. The Jaywalker test bed is a fully active robot meaning the robot has actuation at every joint. This is the most ideal configuration for functionality because the robot does not have to depend on swing dynamics or gravity to walk. The downside to a fully active robot is the energy consumption. Fully active robots, such as Honda's Asimo, require a tremendous amount of energy to operate, which limits the operating time [2]. This energy consumption has been neglected for the short term, and will be addressed in future versions of the Jaywalker's life cycle after the necessary functionality has been achieved. The

assumption is also made that the walker is starting from a double stance position, rather than a single standing stance.

The first control stage is called toe-off. This is when the back heel lifts off the ground and the hip begins to move forward. Toe-off is important because the amount of force applied during the toe-off affects the robots forward velocity, which defines the step length and the swing speed to make a successful step [10]. It is important that the toe-off be accurately controlled because this will allow successful walking over a less strict range of step lengths, as well as helping the robot to operate without fighting the natural swing dynamics of the leg.

The next stage is knee flexion. This stage occurs as the toe from the back foot begins to lift off the ground. Knee flexion is what enables the leg to flex and swing through without contacting the ground [10]. This control stage has been simplified by being in a binary state, either flexion or extension. The amount of flexion in the knee typically does not greatly affect the ability of the robot to walk. In fact, the more flexion the knee has the shorter the lever arm is for the heavy mass at the foot reducing the amount of torque needed at the hip to rotate the leg through. This is contrary to human walking as humans typically only flex the knee enough to clear the leg. The reason is the greater the flexion in the knee also requires an increased amount of energy. To control the amount of flexion the pressure of the air cylinder can be controlled while maintain the binary state of the cylinder.

After the knee flexion stage, the hip can swing the leg through the swing phase. For passive or partial active robots, this phase is controlled solely by the

swing dynamics of the leg. The fully active robot is able to vary this swing with the use of electric motors. Despite the ability to fully control the swing, it is important to avoid fighting the natural dynamics as this greatly increases the amount of torque required at the hip. This is accomplished by running the hip motors at speeds consistent with the natural response of the system. This response can be found by calculating the passive swing dynamics of the system and some trial and error testing to fine tune the performance. A concurrent step is the ankle dorsiflexion. By dorsiflexing the ankle as the hip begins to swing the leg, more space is made for the foot to pass by the ground requiring less overall knee flexion. This also prepares the foot for a future step known as heel strike.

Once the hip has swung the leg, the knee needs to transition to extension [10]. This is the binary opposite state of flexion for the air cylinder. The timing of the knee extension is important to maintain a consistent step length. The straightening of the knee puts the robot in the proper stance for heel strike because the ankle is already dorsiflexed. This is a gait cycle phase rather than a control phase but it is the end of the control phases for taking a proper step. This process is then repeated for the other leg, and that entire process can be repeated to create the robot's gait cycle.

This section describes an open loop control strategy for walking, but closed loop systems and more advanced control algorithms attempt to complete the same phases by monitoring the behavior or sensors of the robot. Chapter 2 will discuss the implementation of the described control structure to achieve the desired gait phases using the Jaywalker test bed.

References

- [1] Short, M., M. J. Pont, et al. (2008). "Assessment of performance and dependability in embedded control systems: Methodology and case study." *Control Engineering Practice* **16**(11): 1293-1307.
- [2] http://world.honda.com/ASIMO/technology/walking_02.html. Date Accessed: 23 March 2010.
- [3] Goscinski, A. and K. Zielinski (1985). "The design of distributed control computer systems." *Computers in Industry* **6**(1): 37-45.
- [4] Xu, S. and J. Bao (2009). "Distributed control of plantwide chemical processes." *Journal of Process Control* **19**(10): 1671-1687.
- [5] Rahkonen, T. (1995). "Distributed industrial control systems -- a critical review regarding openness." *Control Engineering Practice* **3**(8): 1155-1162.
- [6] Neumann, P. (2007). "Communication in industrial automation--What is going on?" *Control Engineering Practice* **15**(11): 1332-1347.
- [7] Cannataro, M., D. Talia, et al. (2002). "Parallel data intensive computing in scientific and commercial applications." *Parallel Computing* **28**(5): 673-704.
- [8] https://computing.llnl.gov/tutorials/parallel_comp/. Barney, Blaise. Date Accessed: 18 November 2010.
- [9] Sunderam, V. S. and G. A. Geist (1999). "Heterogeneous parallel and distributed computing." *Parallel Computing* **25**(13-14): 1699-1721.
- [10] McMahon, Thomas A. *Muscles, Reflexes, and Locomotion*. Princeton University Press. Princeton, New Jersey. 1984.

2. Implementation of the Embedded Control System using the Jaywalker Test Bed

2.1 Jaywalker Mechanical Configuration

The Jaywalker has been developed by the University of Kansas Intelligent Systems and Automation Lab. The Jaywalker consists of a hip and three legs. The two outside legs walk in unison and the middle leg makes the second step creating the 2D biped walking. Figure 2.1.1 shows the Jaywalker standing in the tethered test bed designed for step testing. The test bed described is the second version of the Jaywalker including an upgraded fully active hip from the previous passive hip design [1]. The Independent Hip Drive (iHD) system implemented on the

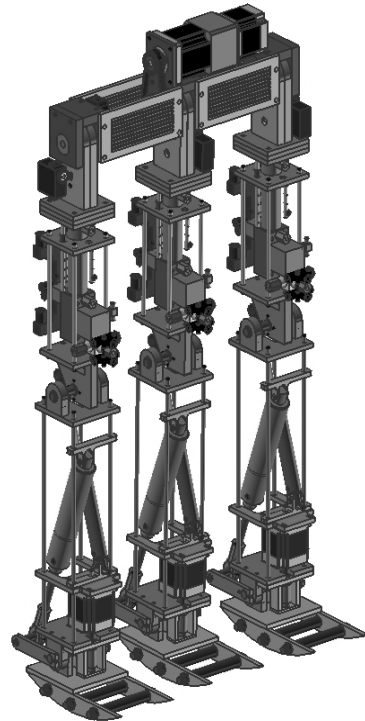


Figure 2.1.1: CAD drawing of the Jaywalker Test bed [1].

Jaywalker consists of two VEXTA PK266M-E2.0B stepper motors [1]. These motors have 0.9 degree step angle, and are capable of producing up to 166 oz –in. or 1.17 Nm of holding torque each. The motors are then connected to a 30:1 gear reduction, which then connects to another 2:1 gear reduction. This creates the 60:1 gear reduction from motor speed to leg swing speed. Figure 2.1.2 shows a CAD drawing of the iHD system. This system allows for each of the outside legs to be controlled independently of each other, which provides some flexibility for gradual turning and fall recovery. The middle leg is fixed to the hip structure and is swung forward by

pushing against the outside legs. When the middle leg is back in the stance phase, this requires less force to swing the middle leg forward rather than swing both outside legs back.

Connected to the iHD are three identical legs. The thigh portion of the leg holds the Leg Extension Guidance System (LEGS). The LEGS system was developed to vary the length of the legs. This system allows for 5 cm (2 in.) of variability in the leg length [1]. This provides another degree of freedom for adjusting the step length to match that of a typical human of the same height. It also has the ability to change length very quickly because of the air actuation, which may provide another means of regaining stability after tripping in the future.

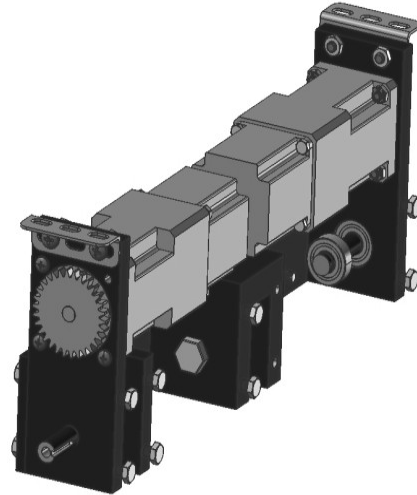


Figure 2.1.2: CAD drawing of the iHD Test bed [1].

The knee joint of the Jaywalker is pinned and is actuated by an air cylinder. As previously discussed, this provides the binary states that allow the knee to be flexed and extended to allow for the leg to swing through and then prepare for heel strike. Figure 2.1.3 is the CAD drawing of the knee system. (formatting and label on next fig)



Figure 2.1.3: CAD drawing of the Jaywalker knee [1].

The ankle joint employs the Hybrid Parallel

Ankle Actuator (HPAA) system [1]. This system makes use of an air actuator and an electric motor to create the necessary torque required for walking. Electric motors provide the necessary fine tune control for the system to make adjustments when the step length is not exactly equivalent, and the toe-off capability is essential to walking on uneven terrain. Electric motors by themselves do not have an ideal power to weight ratio because they are very heavy. Stepper motors are ideal for control because they are relatively easy to control with high precision. Air actuators on the other hand have excellent power to weight ratios, but become very difficult to control precisely [1]. Air actuators typically require several layers of control to effectively control, which in a walking application is not ideal because this requires a significant amount of time at the low level. This reduces the amount of high level control that can be implemented, which can limit the capability to correct for changes in the system. The HPAA system combines the best of both technologies by using air and electric motors in parallel. The air actuator provides the brute force to get the ankle into the estimated force range required for walking. The electric motor is then used to fine tune the output within the operational force band. This provides a simple control logic of a binary state for the air actuator and stepper motor logic to control the entire ankle. This system not only provides a large amount of force at the ankle, but also reduces the weight of using solely electric motors while maintaining the equivalent level of simplicity for control. Figure 2.1.4 provides the CAD drawing of the HPAA system.

Finally, connecting to the ankle is the Jaywalkers curved foot. It has been shown that a curved path is taken by the human foot during our gait [2]. This is more energy efficient than a flat foot requiring a marching style walk. The Jaywalker's foot curvature is modeled after the path followed by the human foot. The problem with a curved foot design is the inability to hold a stance phase. This is the reason for the robot starting in the double stance phase. Ideally, a future foot design will incorporate the standing capabilities of a flat foot with the energy efficiency of a curved foot through the use of powered toes.

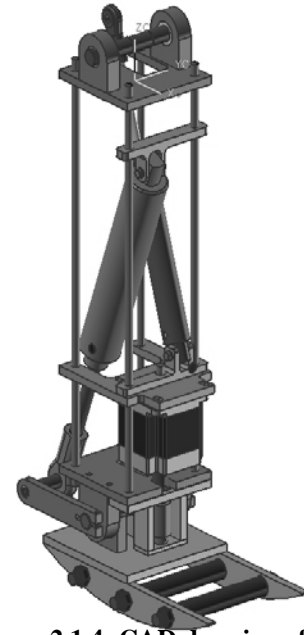


Figure 2.1.4: CAD drawing of the Jaywalker HPA system [1].

2.2 Jaywalker Electronic Hardware

The first step in implementing the proposed control structure was to find an appropriate microcontroller to suite the system requirements. Parallax Inc. has developed a 32 bit microcontroller known as the Propeller chip. It has 32 KB of RAM and ROM on each chip and runs very stable at 100 MHz clock frequency. Each chip incorporates 32 general purpose input/output (GPIO) pins and is typically combined with an additional 256 KB of I²C EEPROM used for program storage. Most importantly the Propeller has eight onboard independent processors denoted as COGS [3]. These cogs are setup in a MIMD configuration, which handles the nontrivial task of implementing a parallel computing structure. The Propeller cogs each have local resources as well as global resources allowing them to communicate

quickly and efficiently. To prevent memory collisions the cogs address the main memory via a round robin style selection called the HUB. The HUB addresses a new cog every two clock cycles meaning an entire cycle through the chip requires a maximum of 16 clock cycles. When a cog is ready to communicate with the Propellers global resources the HUB latches the cog not allowing another address until the current operation is complete. Once, the transmission has complete is proceeds cycling through the cogs. Five Propeller chips are implemented in the system, one for each of the level 2 controllers and one for the level 1 controller. This incorporates a total of 40 processors and 160 GPIO to control the Jaywalker systems.

The power controller interfaces with US Digital MD1 Stepper Motor drivers for each of the five stepper motors powering the Jaywalker. The drivers connecting the hip motors are run at 48 VDC, while the other three powering the ankles are run at 24 VDC. The MD1s are micro-stepping drivers with a 10:1 micro-steps to step ratio. The MD1 driver allows stepper motors to be controlled simply with a pulse width modulation (PWM) signal and a direction pin. The drivers are mounted to a ½ in. piece of aluminum to provide the means to dissipate the heat created by the drivers. This not only helps protect the drivers from overheating, but allows continuous use of the drivers. The second part of the power system is the air cylinders. They interface with Humphrey 310 series solenoid valves. Due to the 12 VDC requirement to change the state of the solenoid, an optoisolated transistor circuit in Appendix B.3 was implemented to drive the solenoids using the Propeller GPIO.

US Digital also supplies the quadrature encoders connected to each of the hip motor shafts. In theory, both legs are given the same commands and should respond accordingly, but taking encoder data on both outside legs allows flexibility to correct for the slight imperfections in the construction of the mechanical parts. The model selected was a high resolution optical rotary encoder US Digital E3. It outputs two quadrature square wave signals and includes the optional index channel to count number of revolutions. This channel is not anticipated to be implemented into the control system as the range of motion of the hip is restricted to less than a full revolution. It is also worth noting due to space constraints, the motors driving the ankles do not provide encoder feedback.

Acceleration is measured in various locations on the Jaywalker using a tri-axis accelerometer sold by Parallax and developed by Hitachi, the H48C Tri-Axis Accelerometer. The H48C is a dual feature sensor because it acts not only as a MEMS rate sensor, but also as a tilt sensor during static conditions. This function allows the sensor to be used to determine an estimated attitude of the Jaywalker, which is essential when the exact starting stance of the Jaywalker becomes unknown. There are seven accelerometers in the following locations: the approximate center of mass of the hip, the approximate center of mass of the thigh of each leg, and the approximate middle of the calf on each leg. These locations provide insight into the overall acceleration of the various limbs of the Jaywalker during movement. The H48C is capable of reading accelerations of $\pm 3g$ with 12 bit ADC resolution. The module sold by Parallax incorporates the MCP3204 four channel ADC, which provides a fast Serial Peripheral Interface (SPI) protocol to read the sensor.

The foot of the Jaywalker is also an important means of timing the walk. Heal and toe sensors provide means to recognize heal strike and the end of toe-off, which trigger subsequent events to begin. Currently, a mechanical reed switch is implemented as the contact sensor. It has proven to be unable to withstand the high impact loads seen during normal gait, and is a topic of discussion currently for improvement. Ideally, the contact sensor implement on the foot is a noncontact style sensor to avoid the cyclic abuse of walking, as several mechanical switches have to proven to fail under these conditions. Future improvements of the foot sensor strategy will be discussed further in the recommendations section.

Visual feedback as the system is operating is provided via an LCD monitor. A VGA connect is implemented to interface the LCD monitor, and provide the real time feedback to observe the system. Fortunately, the Propeller chip incorporates a video configuration register that simplifies the task of driving the VGA signal. The LCD monitor driven by the Propeller also makes it independent of the CPU used to program the chips. This allows the Jaywalker to carry a visual feedback system, and eliminates the need to stream video to an independent CPU making the system entirely embedded in the Jaywalker. In addition to video feedback, data collection is also handled onboard to eliminate the need to stream data to a CPU. An SD card slot is implemented on the Sensor controller as this provides sufficiently fast write speeds and portability to allow offline post processing to be completed on CPUs.

The GUI's input is handled by two PS/2 connectors for a mouse and keyboard. The mouse and keyboard allow the user to interact with the Jaywalkers

menus and create customized test setups. A future revision of the input devices will likely include a wireless mouse and keyboard connection to provide more freedom to the user and reducing the number of peripherals onboard the system.

2.3 Circuit Design and PCB Layout

Implementing the electrical components is done with custom designed PCBs. They were designed using ExpressPCB and ExpressSCH software for layout and schematics respectively. All boards are two layered with no surface mount components. DIP chips were used because the system has a relatively small footprint, space is not a major concern currently, and reduces the soldering equipment required to construct the boards. The controllers are connected in a stackable fashion using header pins and receptors to a main board. The main board connects the Jaywalkers hardware via three 50 pin PC edge connectors and reroutes the pins to their respective controllers. The wiring schematics and PCB layouts can be found in Appendix B and C respectively.

References

- [1] Baker, Bryce. "Development Of A Hybrid Powered 2D Biped Walking Machine Designed For Rough Terrain Locomotion". 2010.
- [2] Adamczyk, P.G., Collins, S.H. and Kuo, A. D., 2006, "The advantages of rolling foot in human walking," *Journal of Experimental Biology*, 209(20), pp. 3953-3963.
- [3] <http://www.parallax.com>. Propeller Chip. Date Accessed: 1 December 2010

3. Jaywalker Software

3.1 Multi-Master Parallel Slave I²C Bus

The Inter-Integrated Circuit (I²C) bus was developed by Philips Semiconductor to provide a two wire interface to a wide range of peripherals. It consists of a Serial Data Line (SDA) and a Serial Clock Line (SCL) [1]. The protocol is a master slave configuration with the ability to support multiple masters on the bus. The I²C bus is not restricted to specific baud rates because the master generates the clock frequency for the bus. There are several standard frequencies ranging from the low speed 10 Kbit/s up to the high speed 3.4 Mbit/s [1]. The I²C bus is also accommodating to level shifts between devices. The standard bus protocol communicates via the following steps:

1. Master sends Start bit
2. Master sends the address of the desired device MSB first with the final bit denoting read/write
3. Slave Acknowledges, confirming address receipt
4. Master transmits/receives byte MSB first
5. Acknowledge from receiver to sender
6. Master issues Stop bit releasing bus

The bus is configured with pull up resistors on each line to avoid floating pin states. Initially both lines are high which serves as the ready state for the bus [1]. The master then sends a start bit by driving the SDA pin low, while the SCL line

remains high. The master then brings the SCL pin to the low state to prepare to generate the clock pulse.

The first byte transmitted is the address of the desired device. The address is predefined by the designer or the manufacturer and serves as a means to isolate the slave the device from other devices on the bus. The address is seven bits long and is sent MSB first. The eighth bit is to denote whether the master is transmitting data to the slave or receiving data from the slave, where zero for the eighth bit is write. All slaves on the bus listen to the address and compare with their own. Assuming the address sent corresponds with a device on the bus, the slave then sends an acknowledge (ACK) bit back to the master [1]. The master then proceeds to transmit or receive data in the same manner as the slave address was sent. It is important to note that the master controls the clock line in the I²C bus even when reading from the slave. Then the device that transmitted information ACKs again to the receiver. This step can be optional when only a single byte of data is allowed to be transmitted per device synchronization. This step is not optional when multiple bytes of data are sent without syncing the devices again. In this situation, an ACK bit is sent in between each byte of data transmitted to help reduce data corruption. When the final ACK is received, the master then issues a stop bit by first releasing the SCL line and then the SDA line signaling the end of the communication.

The I²C bus provides many benefits for the proposed distributed controller configuration including the relatively low number of GPIO required to communicate. Each controller in the system has the potential to be a master so to successfully

implement an I²C bus the multimaster setup is required. Luckily, this setup is included in the standard, but does present an inherent problem called arbitration [1]. Arbitration occurs when two masters attempt to communicate simultaneously. It is important that this situation is detected and one master's communication is stopped. Fortunately, this is a relatively easy problem to solve in theory. As each of the bits are sent arbitration can be checked by ensuring the bit is correct. If two masters are sending data on the line at some point their data will differ, and the device sending a one bit will recognize the line is low. This master then surrenders the bus to the other master and waits for a stop bit to attempt another transmission. Typically it is not robust enough to only check arbitration on the address byte as two masters may wish to communicate to the same slave, and the arbitration would not be recognized until the data byte is being sent. This process does slow the performance of the bus but is necessary in a multi-master configuration to avoid collisions on the bus.

The purpose of the proposed Multi-Master Parallel Slave (MMPS) I²C bus is to retain the advantages of the standard multi-master setup while increasing the data throughput rate. The first major difference between the MMPS I²C bus and the standard is the use of a command byte rather than an address byte. The first byte transmitted with the MMPS I²C bus is a seven bit command with the eighth bit still denoting reading or writing. The method lets slaves on the bus listen for a particular command that applies to it. The commands are predefined by the user like the slave addresses, but the difference is the command corresponds to a description of what is being sent rather than which device it is being sent to. For example, to send an encoder reading from one controller to another the standard bus would first send the

address of the device and then send the sensor reading. The MMPS I²C bus would send a command byte that corresponds to an encoder reading selected arbitrarily by the user and then sends the sensor reading. The advantage of using this type of system is the ability to send the same piece of information to multiple slaves simultaneously. For example, the encoder reading may need to go from the power controller to the sensor controller for data acquisition and the GUI controller to be displayed. The standard bus protocol would require two communications to accomplish this task, but the MMPS I²C bus only requires one.

There are a couple problems that arise that need to be dealt with to successfully implement this protocol. The first is synonymous with the standard I²C bus with multi-master capability. The MMPS I²C bus handles master device arbitration in the same manner as the standard bus. A new arbitration problem arises by allowing multiple slaves to listen to the commands, and it occurs when the slave devices are ACKing after receiving bytes. An intelligent means to control the order of acknowledging is required because multiple slaves could ACK simultaneously and it would only be seen as a single ACK by the master, which would trigger a false error. There are two proposed solutions to this problem: High Priority ACK (HPA) and ACK In Turn (AIT). The AIT method is the more robust of the two because it requires an ACK from all slaves that are receiving the data. AIT works by predefining an ACK order for the slave devices for each command. After the command or data has been received the slaves then ACK in the predefined order, and simply subtracting the numbers of ACKs that has occurred minus the order number of the slaves denotes when it is a slaves turn to ACK. The master then keeps

a total count to ensure all slaves have successfully received the information. This order can be arbitrary so each new device added to the bus can be added to the ACK order for the commands they would receive. This method reduces the data corruption and provides a means to error flag a device that is not operating properly.

The second solution is the HPA method. This is done by making the assumption that of the slaves one device is significantly more important than the others. For example, it is significantly more important that the encoder reading in the previously examples is successfully read by the sensor controller than the GUI controller because the GUI system is for visual feedback but not for control or analysis. The HPA method only requires a single ACK from the highest priority device. In the example, the sensor controller would be the only device that would ACK back to the master. It is then assumed that either the data was also successfully received by the other slaves or it is not detrimental to the system that the data was corrupted. This method serves as a means to provide an optimization over the AIT method by reducing the total number of ACKs, but is less robust at ensuring accurate data transmission.

The MMPS I²C protocol provides the opportunity for significant data throughput increases if redundant information is being sent across the bus. Another means to optimize the transfer rate is to relax the arbitration checking by the master as well. Depending on the system and the commands that are predefined it is possible to only check arbitration on the command byte thereby speeding up the communication. For example, the only device in this system that can send an encoder

reading is the power controller so we do not have to worry about another master on the bus sending the same command because no other device can take that measurement. This shortcut works for the proposed control system because the predefined commands are unique for each controller; therefore, two masters would never send the same command byte. This is not true in the generic sense, and therefore this optimization can only be performed when it applies to the individual system.

Despite the speed advantages it is important the MMPS I²C bus communicates with standard I²C devices, such as EEPROM, seamlessly. Luckily this is a trivial problem because the predefined command in the MMPS implementation only needs to correspond to the slave address of the device, and the slave device will not recognize the difference other than the slave arbitration. It will be important to handle the slave arbitration using the HPA method in that situation because the standard I²C designed device will not be capable of counting ACKs or waiting for a turn.

The MMPS I²C bus was successfully implemented on the Parallax Propeller chip by writing low level bit-banged routines. The data transfer rates were not able to be increased up beyond the equivalent low speed operation of the standard I²C bus. This is most likely a byproduct of writing the bit-banged routines in a high level language. Speed increases would most likely be attained by implementing the routines in assembly.

3.2 Emergency Bus

The Jaywalker is comprised of custom parts and relatively expensive hardware, so it is very important to protect hardware from damage during failing conditions. The Emergency Bus (EBUS) is a One-Wire Serial bus that is used strictly to monitor events of interest and shut down the power systems. The One-Wire Serial bus was developed by Dallas Semiconductor and behaves very similarly in concept to the I²C bus [2]. It uses a start bit to synchronize the devices, sends a device address to identify the correct slave device, and then sends the information. The master also receives ACK bits from the slave device in the One-Wire Serial bus. The major difference is the One-Wire bus uses one pin for data transfer and the clock. As a byproduct, the data transfer rates are typically low comparatively because longer clock pulses are typically used to ensure devices with different clock frequencies do not miss data [2]. The low transfer rate is not a disadvantage for the Jaywalker because it only needs to monitor the wire for an event of interest. Events of interest include failing conditions in the hardware or software that could cause harm to the system. The EBUS is configured with a pull down resistor, so low state signifies an operational system. The line is driven high during a failure and power is disconnected to the power controller. In addition to removing hardware power, all processors on all chips are stopped unless the processor triggered the failure condition. The EBUS then outputs an error code to the LCD monitor.

The first and most important trigger for the EBUS is the manual button. This button is pressed by the user when failing conditions are directly observed. A variety

of software events can also trigger the EBUS, such as a motor feedback timing out. This would occur if an encoder was failing or a motor controller was not responding. Each separate event can trip a user designated code to aid the troubleshooter in finding the source of the failure. The EBUS has the ability to not only protect essential hardware, but provides a great resource during fall testing to identify mechanical bottlenecks of the system. The EBUS will allow future researchers to test the limits of the Jaywalker and provide feedback as to how and why the system failed given known conditions. This information will be vital in the efforts to improve the mechanical robustness of the system.

3.3 Jaywalker GUI

The Jaywalker GUI is designed to allow the user to test all systems of the Jaywalker together and independently to eliminate the need to reprogram the chips simply for new test plans. It uses simplistic graphics with 1024 x 768 pixel resolution. Currently the GUI is broken down into four menus including: Walk, System Check, Manual, and Debug.

The walk menu is used to specify parameters of continuous walking. It only requires the user to input which leg will take the first step, how many steps to take, and a start button to begin the walking cycle. The walk menu makes the assumption that it uses the current algorithm in the main controller. It also does a onetime syncing of the chips to ensure communication lines are operating properly before walking. The data acquisition is then started and finally the power controller is

started and commands are sent to walk. This menu is designed to provide very trivial interface when simple walking test plans are desired.

The next menu is the systems check. This is a menu that is designed to verify proper operation of all parts of the Jaywalker. The intention of the systems check menu is to provide a fast means of troubleshooting the mechanical and electrical hardware after a rebuild or rewire. The systems check allows the user to select specific slots which correspond to DB-25 connectors on the robot to test a small group of systems or all slots. There is an output menu to show progress of the systems check and to display error flags. An input line is also available to allow the user to test an individual component in a single slot by prompting the user with a test menu in the output window. Finally, the start button begins the tests and switches to a stop button if the user wishes to exit the test before completion.

The manual menu is used to set preferences for a low level test. It provides lower level control over the individual systems and allows independent control over the power and sensor systems. This allows the user to perform very controlled tests before implementing a full walk. The manual menu also allows the user to disable features that are not required for a specific test. All sensors default to on and power systems default off. The hip motors also default to opposite directions because this rotates the outside legs in the same direction because of the reversed orientation of the motors relative to each other. It is important to note that the manual menu does not include a start button. This is because it is simply intended to set the preferences for a

test, but to actually run the desired commands the user must switch to the debug menu.

The debug menu is primarily a set of output menus. The start button on the debug menu first grabs the preferences set by the manual menu and then executes the commands that correspond. The debug menu has separate output menus designed for the component listed in the title to provide feedback as the system is running. This also outputs any errors that occur during operation. The manual and debug menus are structured in this manner to allow the user to switch to the output menus before the system is running. This allows the user to never run tests “blindly”, which is important as this is primarily intended to testing new functionality.

Future revisions of the menu structure will likely incorporate an easier means to program the controllers as code changes are made. It will also incorporate real-time plotting for various sensor readings.

References

- [1] Eisenreich, D. and B. DeMuth (2002). The I2C Bus. Designing Embedded Internet Devices. Burlington, Newnes: 433-466.
- [2] Eisenreich, D. and B. DeMuth (2002). 1-Wire Basics for TINI. Designing Embedded Internet Devices. Burlington, Newnes: 345-431.

4. Jaywalker Testing

4.1 Motor Ramping

The first test the new electronic system needed to pass was the ability to ramp the stepper motors acceleration. The micro-stepping drives used to drive the motors are not capable of handling acceleration ramping from a cold start. This means to reach the higher acceleration required by the hip swing the motor speed needs to be stepped up rather than instantly accelerated. The key to accomplishing this task successfully with minimal motor jitter is to have the acceleration steps occur without breaking the pulsing order of the motor phases. This keeps the motor operation smooth and allows it to accelerate until the optimal velocity is reached. Another consideration for this testing was to ramp the motor as quickly as possible, because the motor movements are relatively fast. If the motors are ramped too slowly, they would reach their final position before reaching their final velocity.

Testing showed that the stepper motors were capable of accelerating to a velocity of 5 rpm under load without ramping. The previous commercial motor controller; however was capable of reaching velocities up to 17 rpm. The new control system was found to require two ramping steps to reach 17 rpm, and was able to do so 0.25 seconds. This ramping fell within the hip swing time frame of 0.5 seconds. Future motor tests will likely result in a reduced ramping time by a more robust means of handling the phase steps during ramp transitions.

4.2 Range of Motion

The Jaywalker has the fully active iHD implement in addition to the new control system, so it was necessary to test the hip range of motion to insure it was sufficient for walking. The first test was to find the range of motion of each outside leg with the leg bent to simulate hip swing conditions. The control system was able to drive the iHD to 50 degrees forward from standing and the same 50 degrees backward from standing. It is believed that further range of motion is possible, but this test was done while the Jaywalker was in the air. This did not give the system resistance to push against, such as when the feet are on the ground. This test was sufficient to prove necessary range of motion needed to take steps, so a ground test was not necessary.

Next, the same test was run with the leg actuators active keeping the legs straight. This simulated the highest torque possible by the system. The system began to back drive the motors due to insufficient holding torque after 45 degrees in each direction. This test defined the operating limits of the Jaywalker with straight legs.

4.3 Data Acquisition

A major problem fixed with the new electronic system was the data acquisition throughput rates. The previous system was providing one axis on two accelerometers as a rate around 20 Hz. Because the Jaywalker's step only takes around half a second, the system was only able to take 10 data points for each step. This caused the controller to have problems finding maximums during the stepping.

The new data acquisition implemented on the sensor controller was able to greatly improve the data throughput rates. It is capable of reading all three axes of the tri-axis accelerometers as well as reading seven sensors rather than two. It is able to then stream this data to the SD card at rates between 400 - 600 Hz. The system is capable of producing even higher throughput rates by tuning the buffer size and optimizing communication timing with the SD card. This system provides a much more complete sensor coverage of the Jaywalker, in addition to increasing the number of data points per step by 20 – 30 times the previous system. This makes the accelerometers a viable option to be used for control schemes in the future.

4.4 Step Testing

The final test to perform after successfully passing the previous tests was to take an open loop step. The open loop control algorithm is very simplistic and relies heavily on accurate step length repeatability to tune the controller. Because of this dependence on the ability to set the initial step length accurately, the success rate was only around 15%. It was found to be successful at a step length around 7.5”, which was increased from 5 ¾” performed by the previous control system. The system was tuned by adjusting the timing in the main controller, which implemented a predefined order of commands. The step taken with the middle leg is shown with still frame images in Appendix E.

Conclusion and Recommendations

The Jaywalker is a fully active 2D biped walking robot intended to study locomotion on uneven terrain. An embedded system was chosen to provide the control structure for the Jaywalker due to the numerous advantages over the previous PC control system. Walking robots require several simultaneous tasks to occur for successful operation. Distributing the control problem helps reduce the complexity of the individual pieces and increases the overall performance of the system. Parallelizing the system also provides much needed computational efficiency. A two level control system was developed by combining these concepts that provides robustness and algorithmic flexibility.

The system was then diagramed and laid out for custom circuit boards to keep the footprint compact. The controllers were designed as stackable modules that shared a common connector board, which also connected the Jaywalker hardware via edge connectors. The five total controllers were implemented on Parallax Propeller chips and communicate via an MMPS I²C bus. The MMPS I²C bus is a multi-master configuration that is command based rather than slave address based. This allows multiple slaves to listen to the information simultaneously, reducing the total number of transmissions when redundant information is sent.

The Jaywalker incorporates a GUI that allows the user to run test ranging from a single individual component to a programmed walk. The GUI displayed using an LCD monitor via a VGA signal. The user provides inputs using the mouse and keyboard.

The embedded control has shown considerable success by taking steps using simple open loop control. The success of the open loop system is heavily dependent on the starting step length, but the open loop control is essential for testing the Jaywalker's individual components to verify mechanical integrity before attempting continuous walking. The step length was increased from 5 1/2" to 7 1/2" with the embedded system, and has the potential to significantly increase the step length further. Overall, the system has performed flawlessly despite the simplicity of the current control strategy.

Recommendations for future improvements to the Jaywalker include:

- Closed loop control with encoder feedback to allow predictable steps for uneven terrain testing
- Intelligent foot to provide more information regarding the foot's orientation with the ground. This will give the information needed to make relations between stability and the behavior of the system.
- Dynamic plots of sensor data in GUI. Plots will give the user real time trends rather than a raw data dump, which is beneficial for relating the movement of the system to the sensor readings.
- Low level I²C functions written in Assembly. This will allow the communication rate to be stable at 400 KHz, making it equivalent to the high speed I²C rates.

Appendix A: Jaywalker Control Flowchart

Appendix A contains flowchart for the Jaywalker test bed. The boxes represent software programs, and the circles represent hardware and peripherals. The boxes with controller in the title are the five Propeller chips.

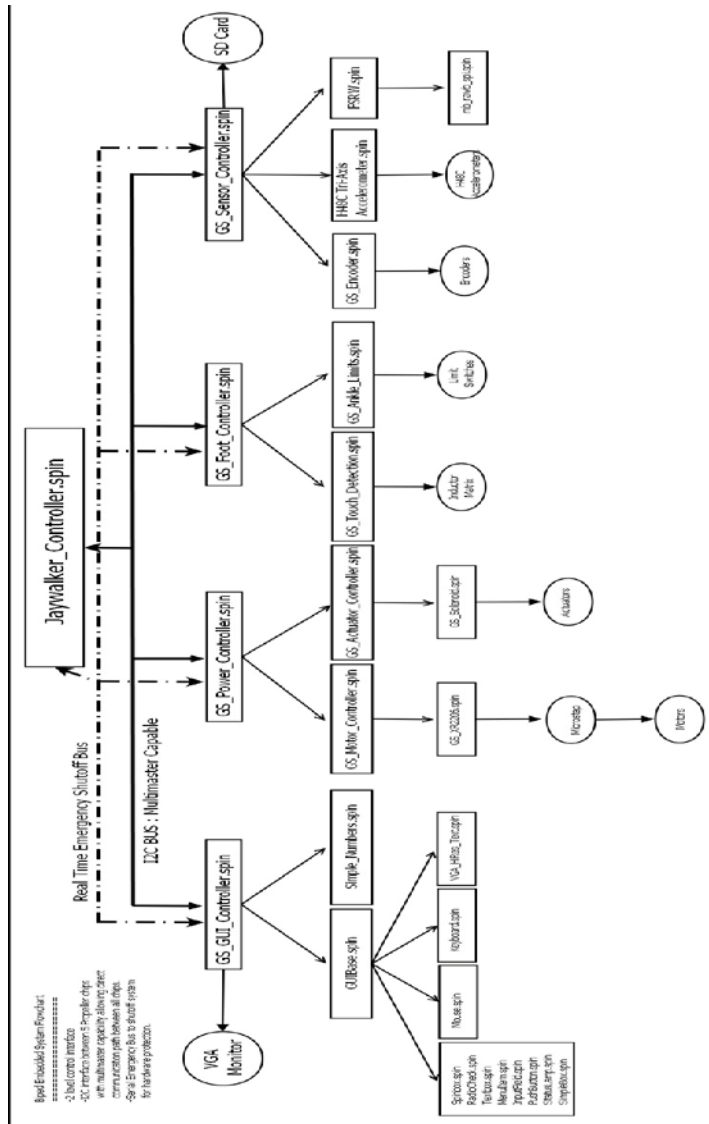


Figure A.1: Jaywalker Control System Flowchart

Appendix B: Jaywalker Wiring Schematics

Appendix B contains the wiring schematics for each of the controllers, the connector board, and the circuits to drive the power systems.

B.1 Level 1 Main Controller Wiring Diagram

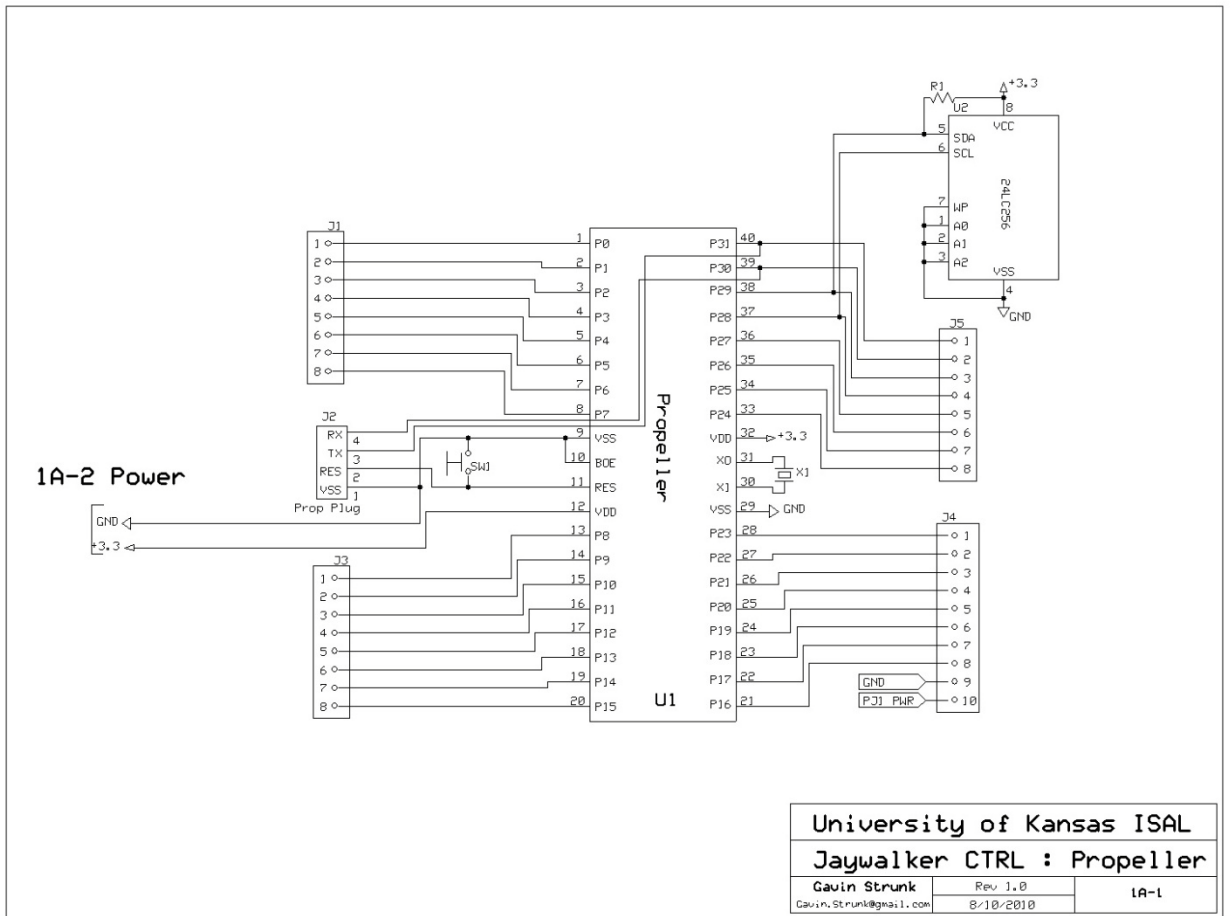


Figure B.1.1: Propeller wiring diagram with EEPROM and connectors corresponding to GUI Controller connectors.

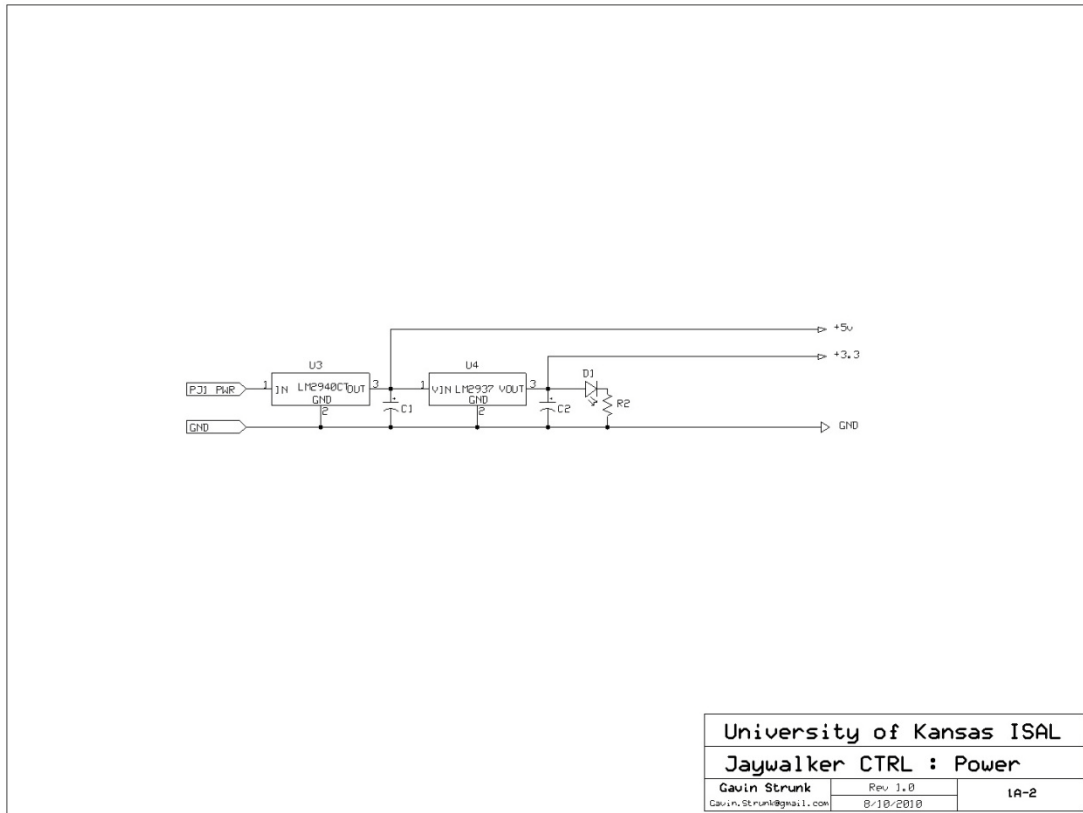


Figure B.1.2: Regulator power source provided +3.3 VDC and +5 VDC for the Main Controller.

B.2: Level 2 GUI Controller Wiring Diagram

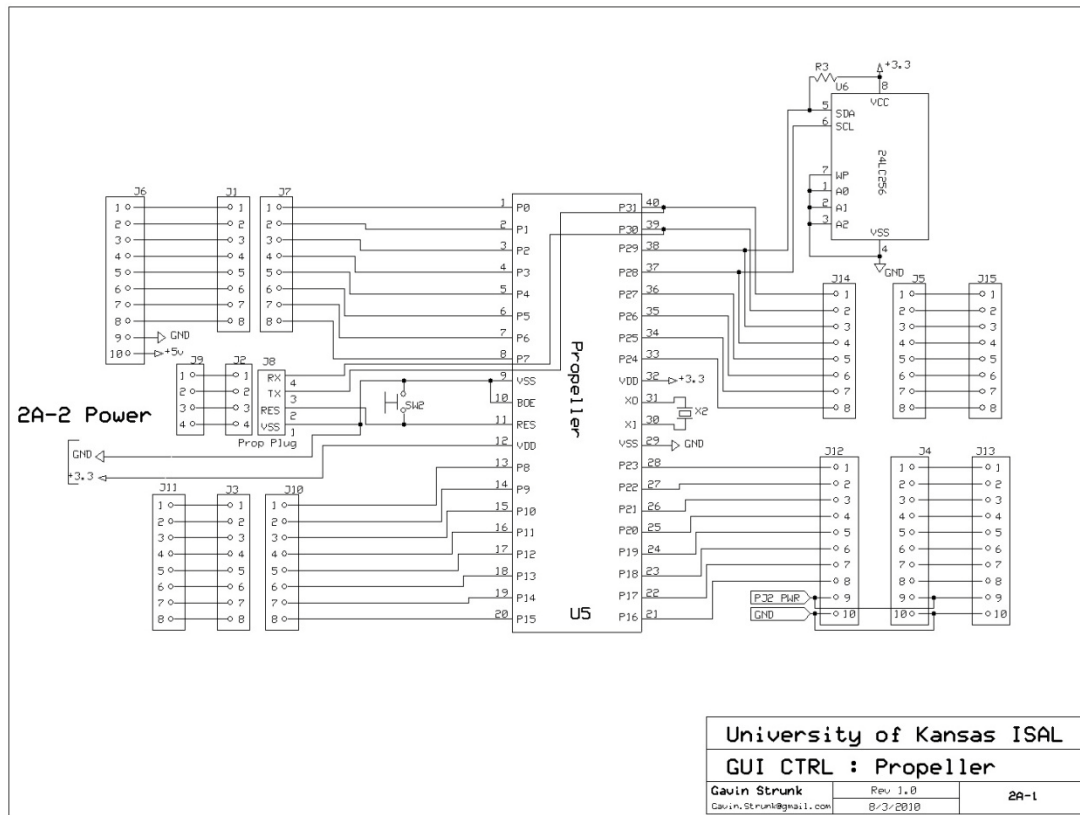


Figure B.2.1: Propeller wiring diagram with EEPROM and connectors corresponding to the connector board.

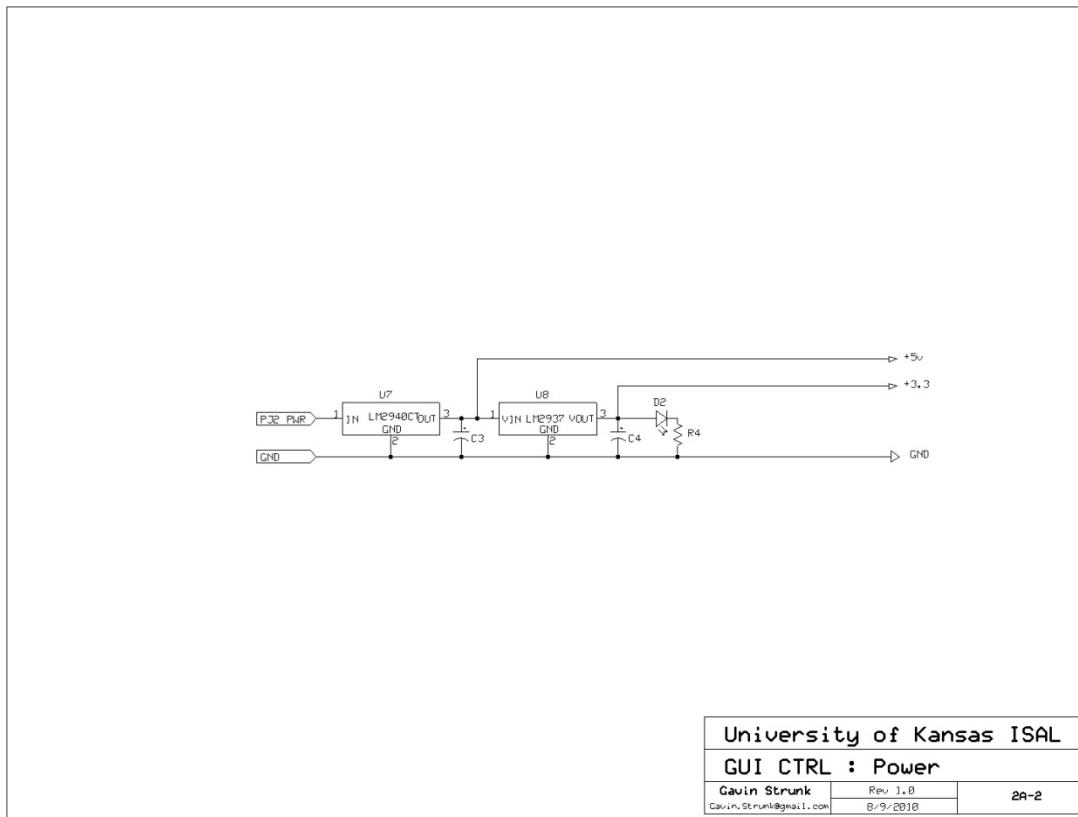


Figure B.2.2: Regulator power source provided +3.3 VDC and +5 VDC for the GUI Controller.

B.3: Level 2 Power Controller Wiring Diagram

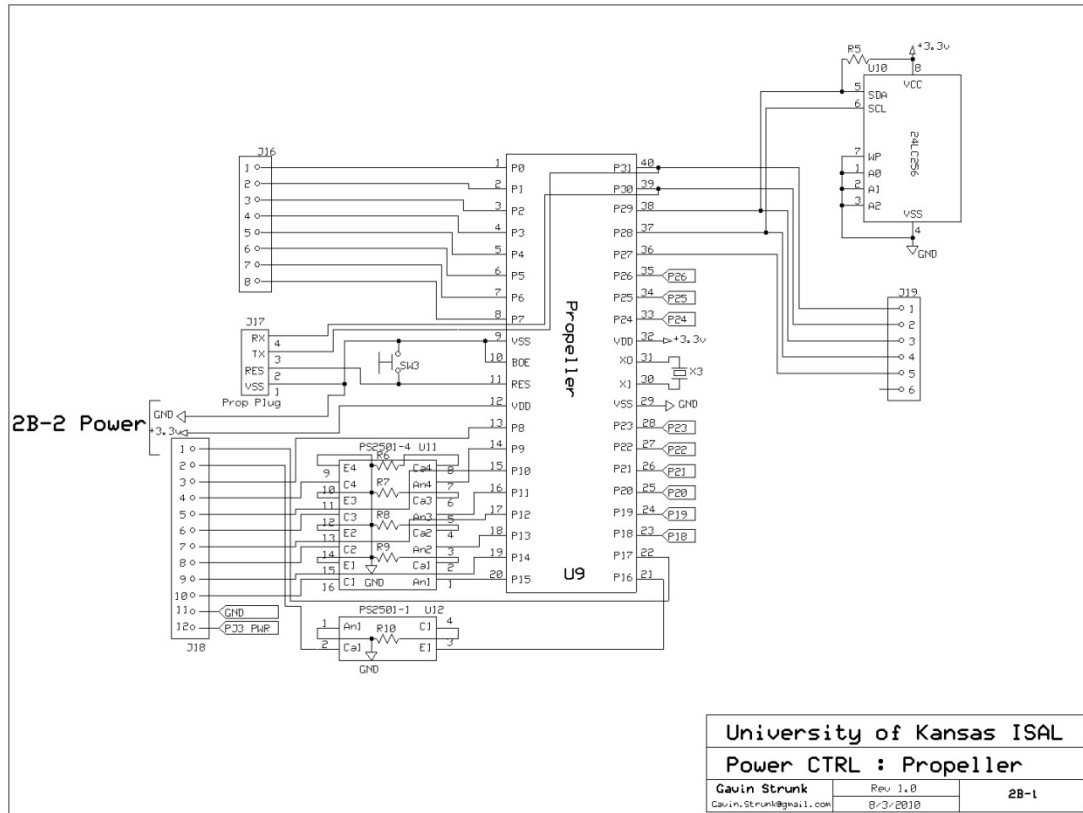


Figure B.3.1: Propeller wiring diagram with EEPROM, optoisolator circuit for direction pins on MD1 motor drivers, and connectors corresponding the connecting board.

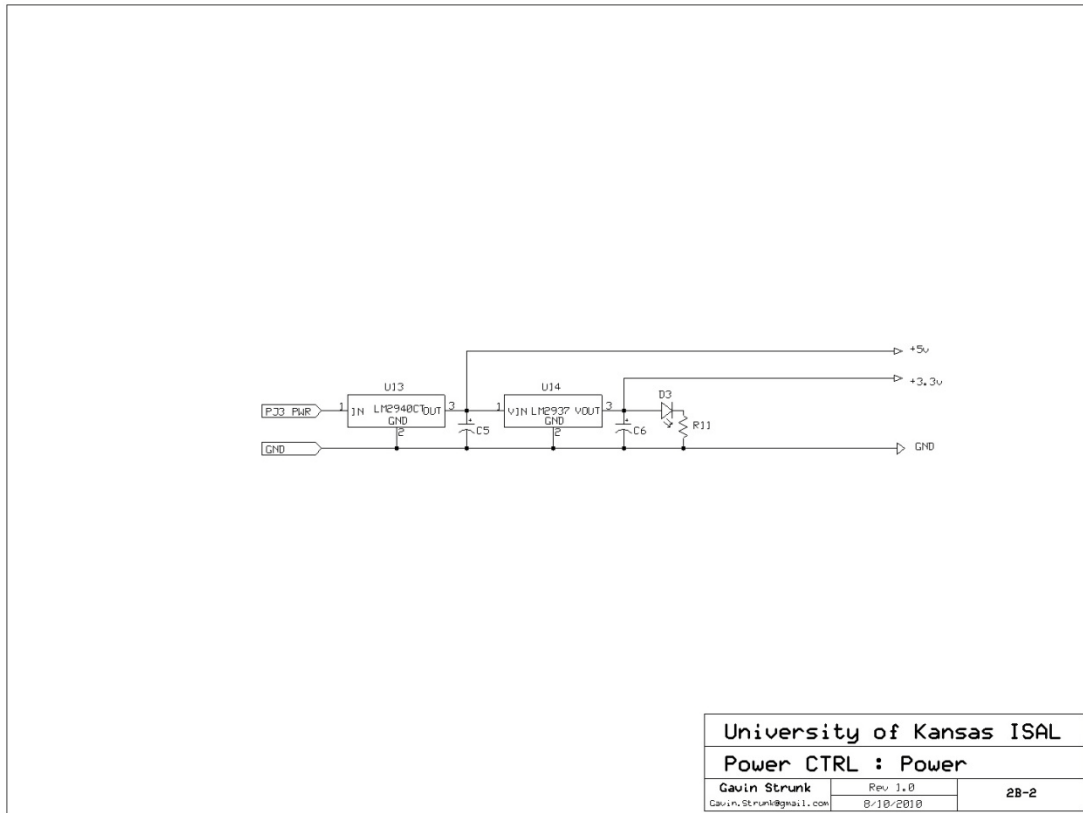


Figure B.3.2: Regulator power source provided +3.3 VDC and +5 VDC for the Power Controller.

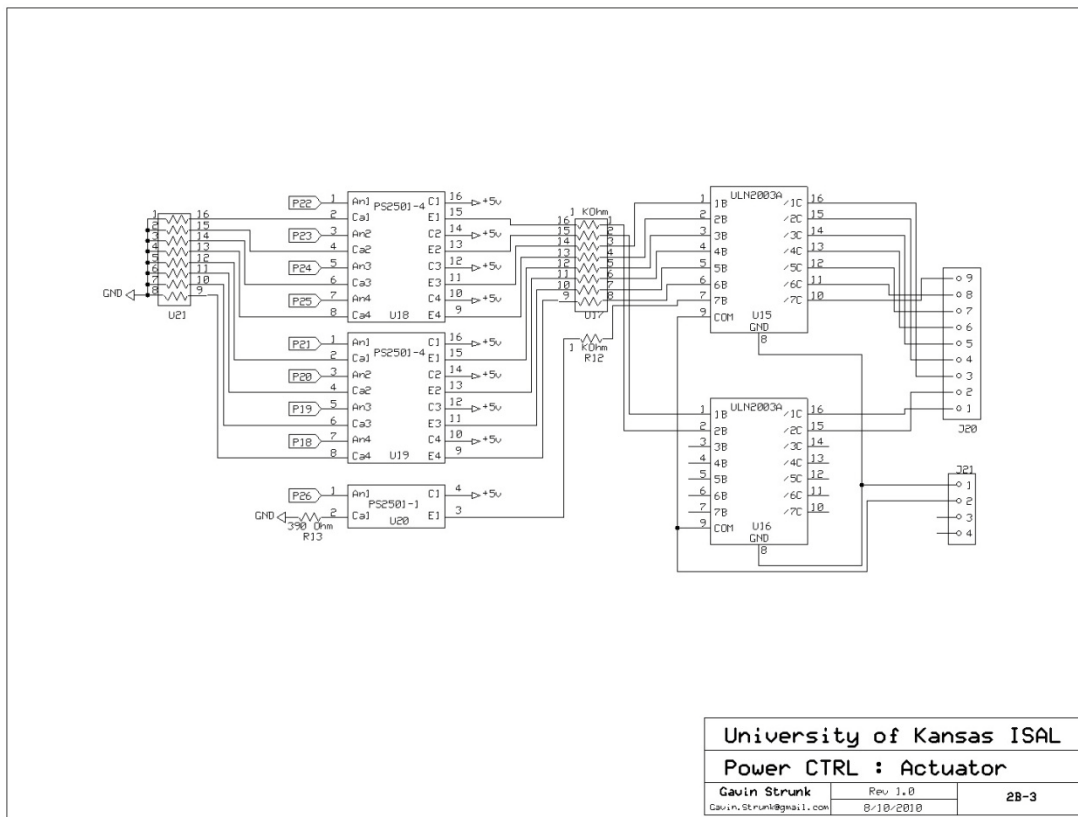


Figure B.3.3: Optoisolated transistor circuit to drive the +24 VDC solenoids for the air cylinders.

The schematic diagram illustrates the internal wiring of the Propeller 2C-1 module. It features three main pin headers: J22 (pins 1-8), J23 (pins 1-4), and J24 (pins 1-8). The Propeller IC (P31-P39) is connected to J22 and J24. The U22 IC (P32, P4R) is connected to J24 and a separate P32/P4R header. The U23 IC (24C32) is connected to J23 and a separate U23 header. The 2C-2 Power header (GND, +3.3) is connected to the Propeller's VDD and VSS pins. The U22 IC is also connected to GND and +3.3. The U23 IC is connected to VCC and VSS. The diagram shows a complex network of connections between these components, including a pull-up resistor on the Propeller's BOE pin.

54

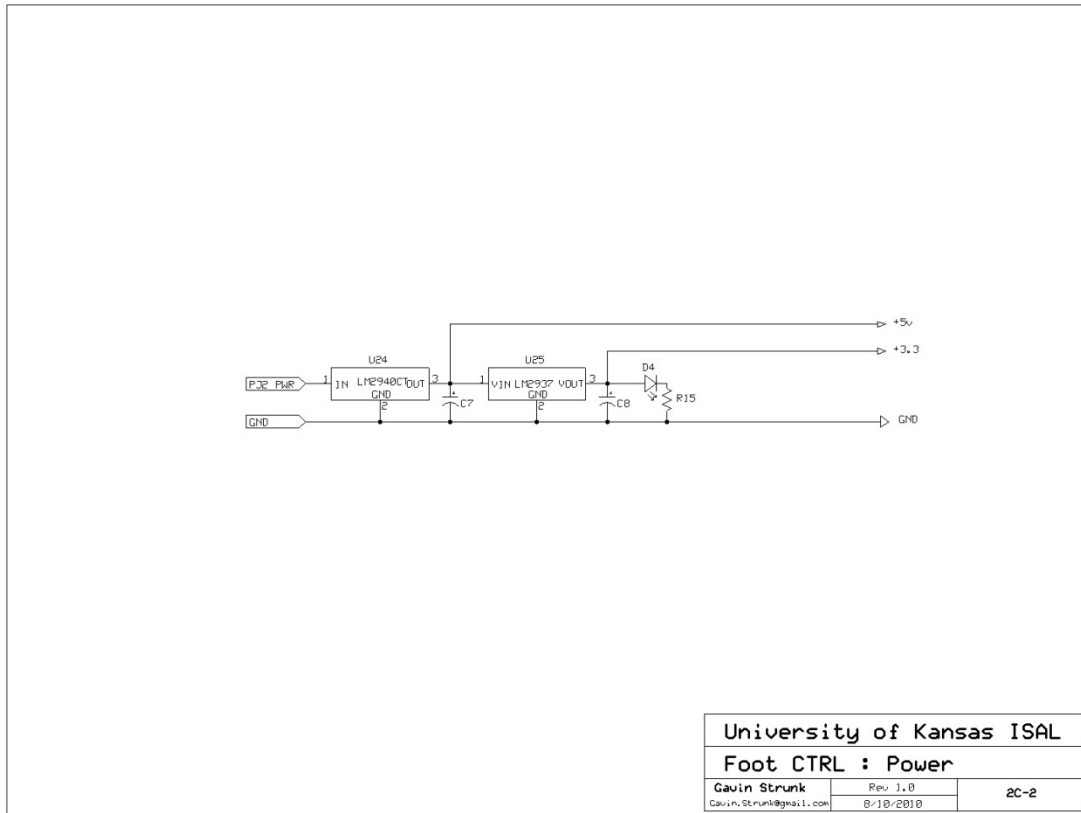


Figure B.4.2: Regulator power source provided +3.3 VDC and +5 VDC for the Foot Controller.

B.5: Level 2 Sensor Controller Wiring Diagram

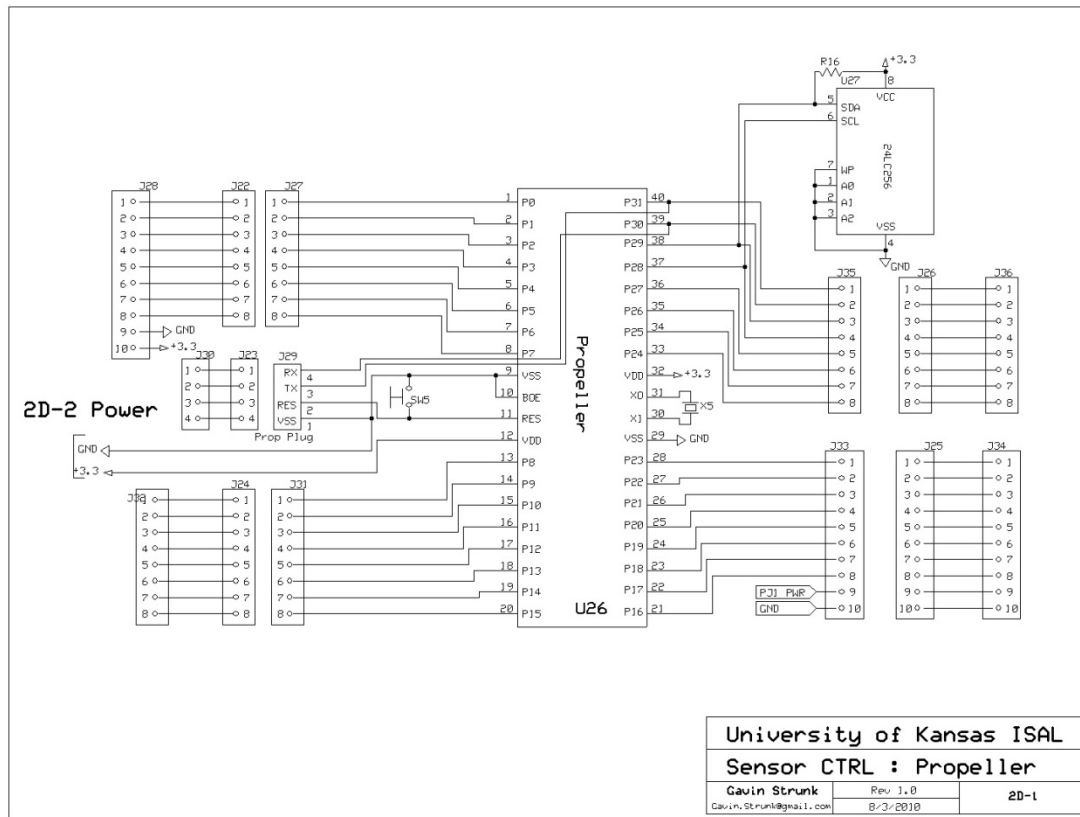


Figure B.5.1: Propeller wiring diagram with EEPROM and connectors corresponding to the connector board.

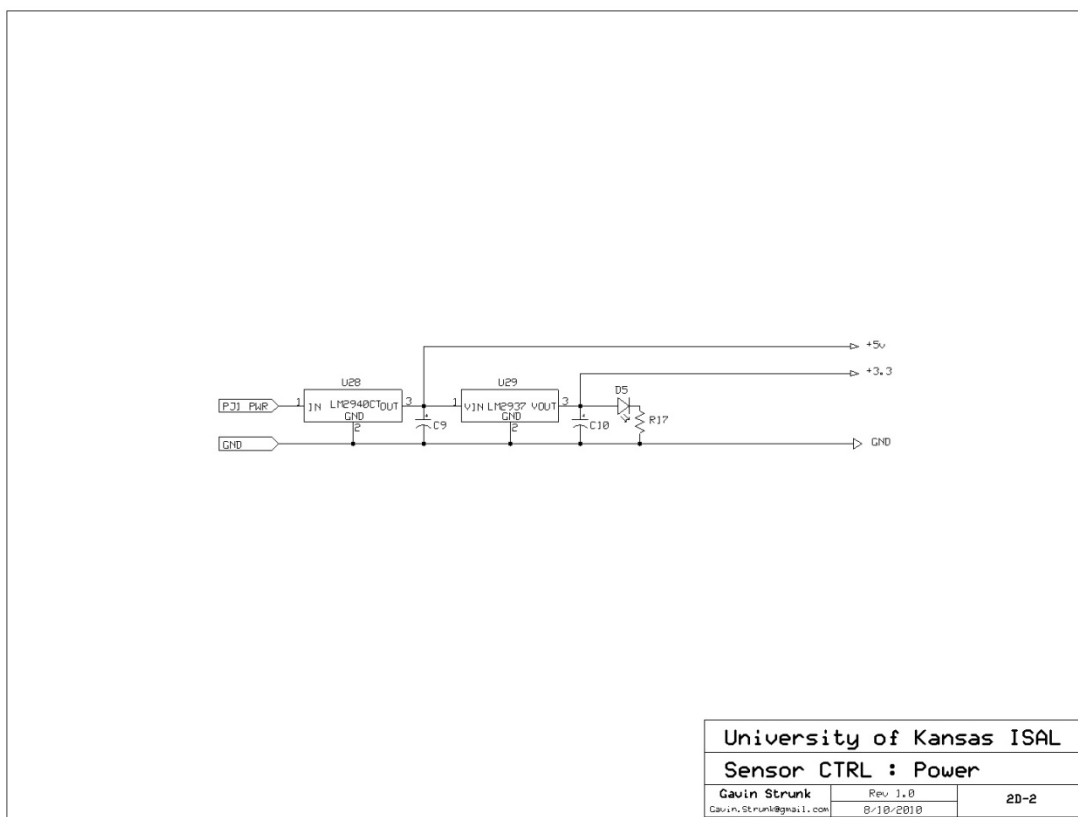


Figure B.5.2: Regulator power source provided +3.3 VDC and +5 VDC for the Sensor Controller.

B.6: Connector Board Wiring Diagram

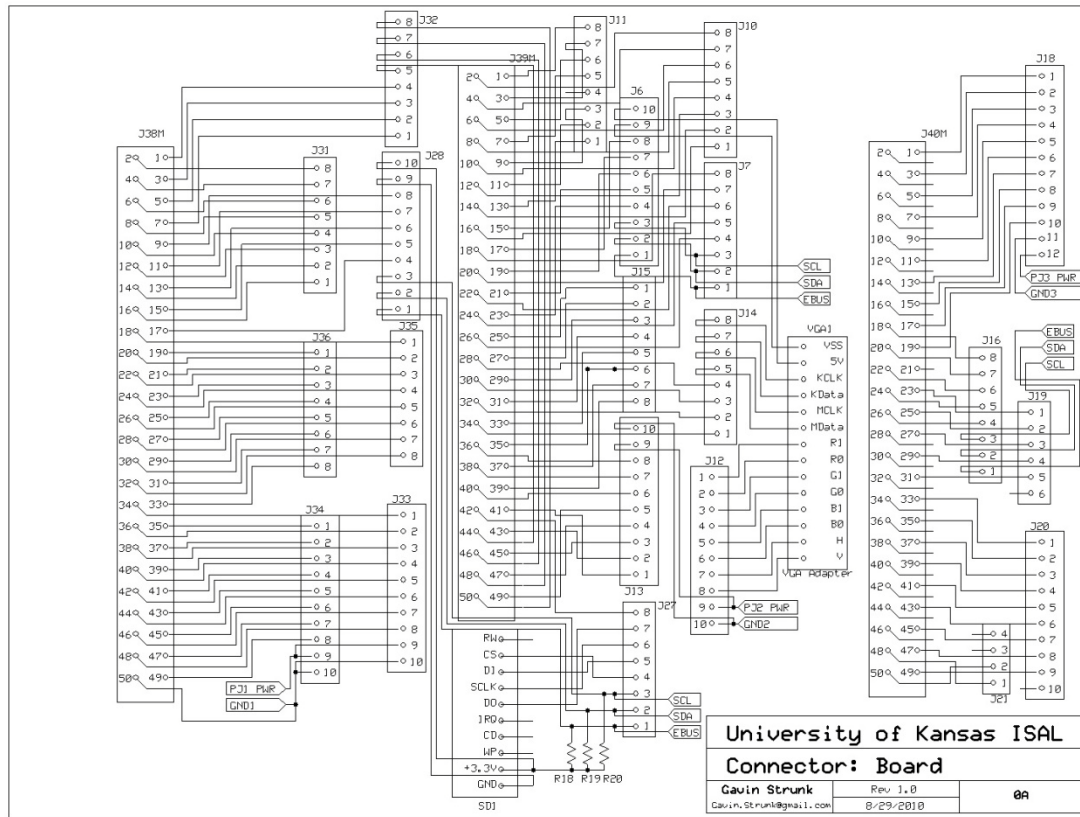


Figure B.6.1: Wiring diagram for the connector board including slots for a VGA adapter, SD card adapter, and receptors for the GUI, Sensor, and Power Controllers.

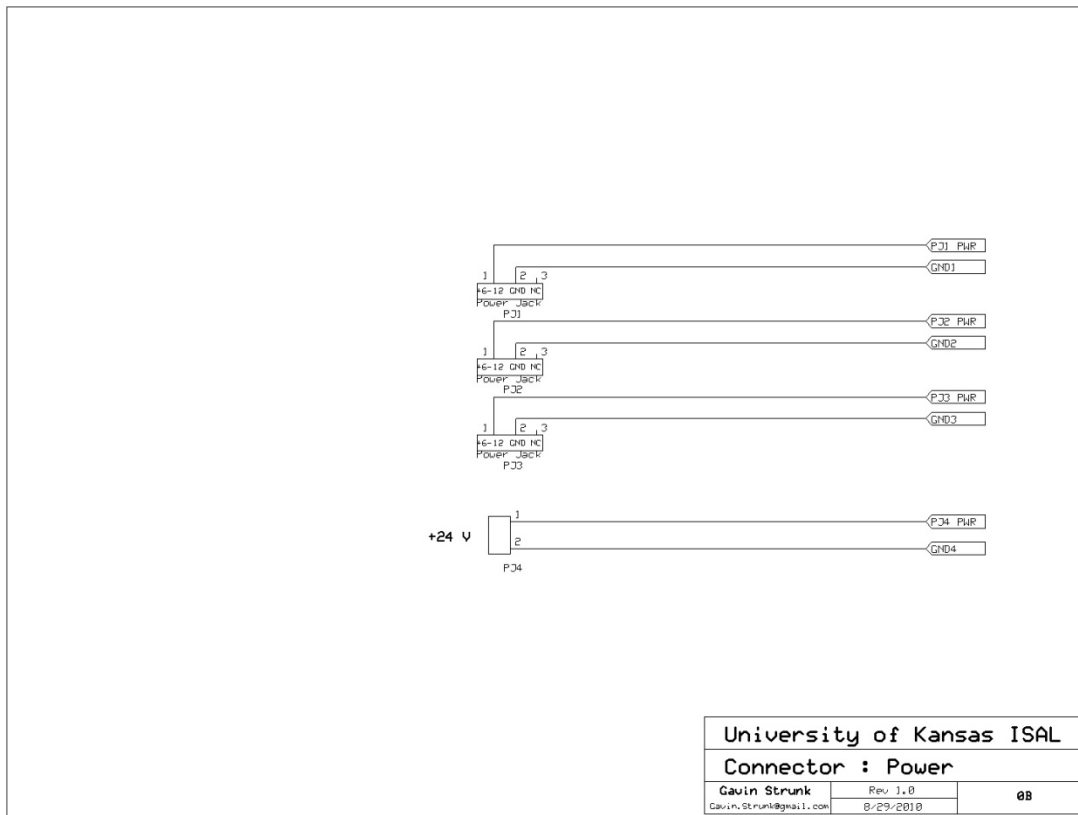


Figure B.6.2: Regulated power source providing +3.3 VDC, +5 VDC, and +24 VDC

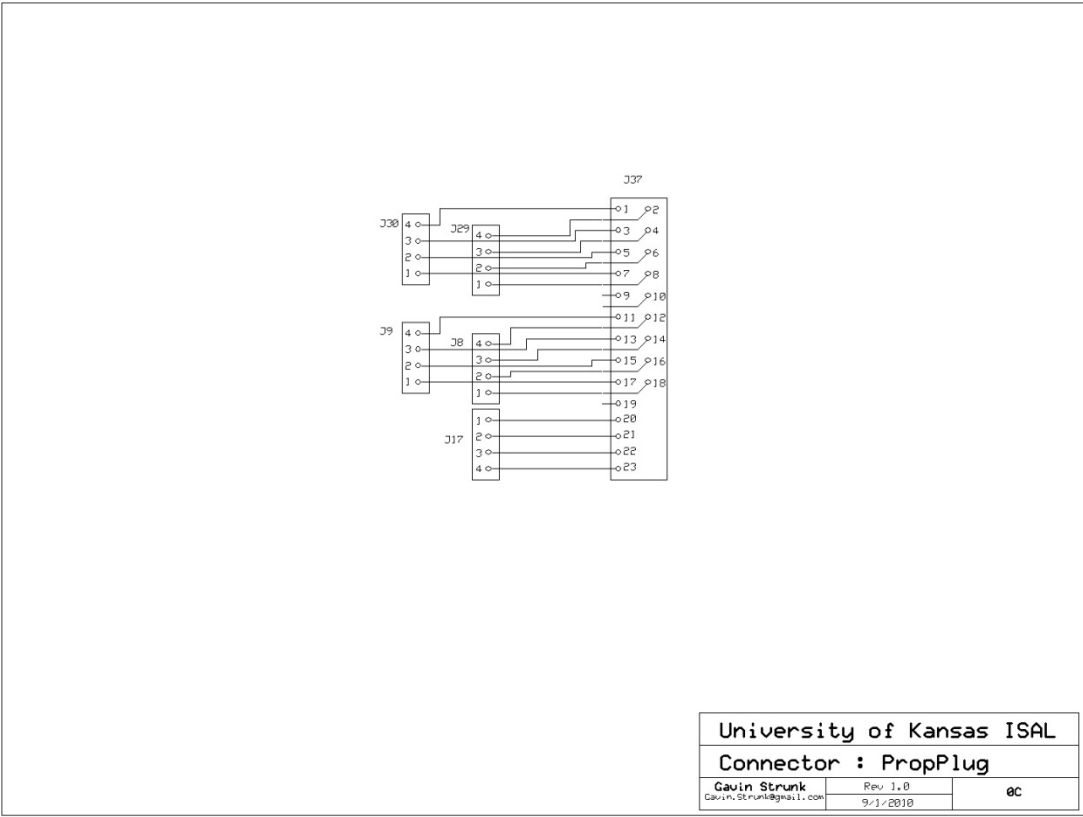


Figure B.6.3: Connector for PropPlug to program each of the five controllers.

B.7: Jaywalker DB-25 and Round Wiring Diagrams

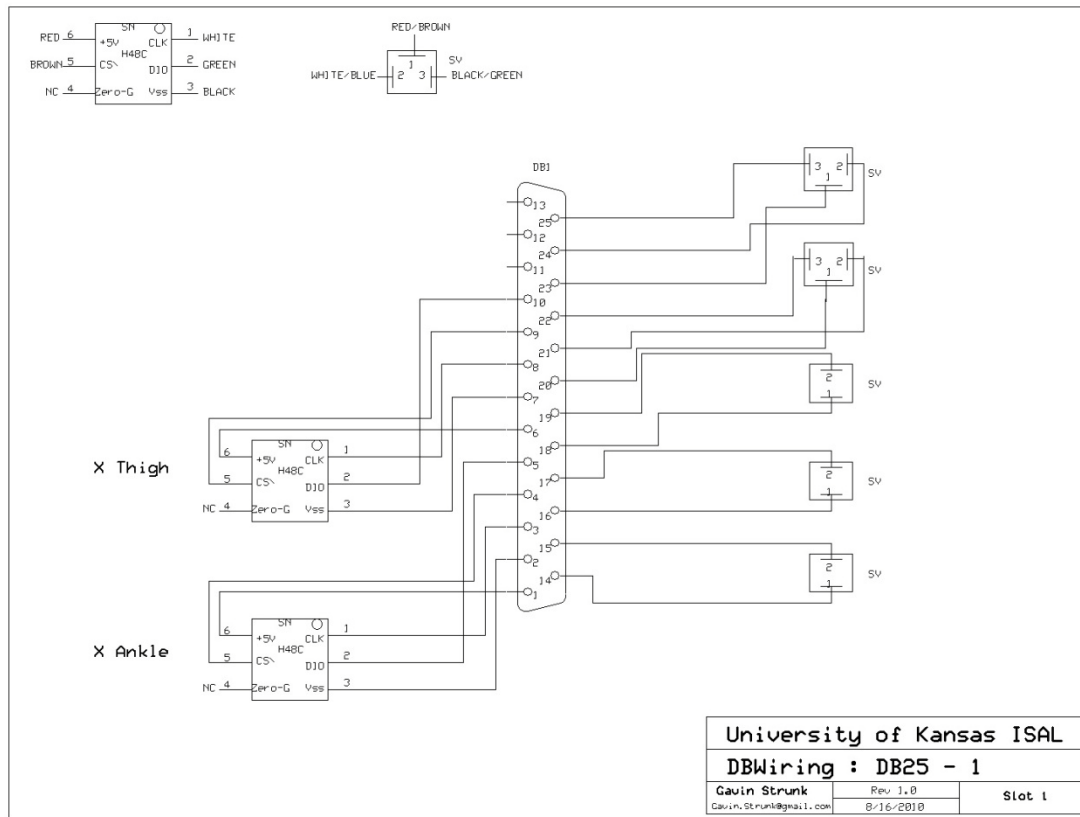


Figure B.7.1: DB25 connector in slot 1.

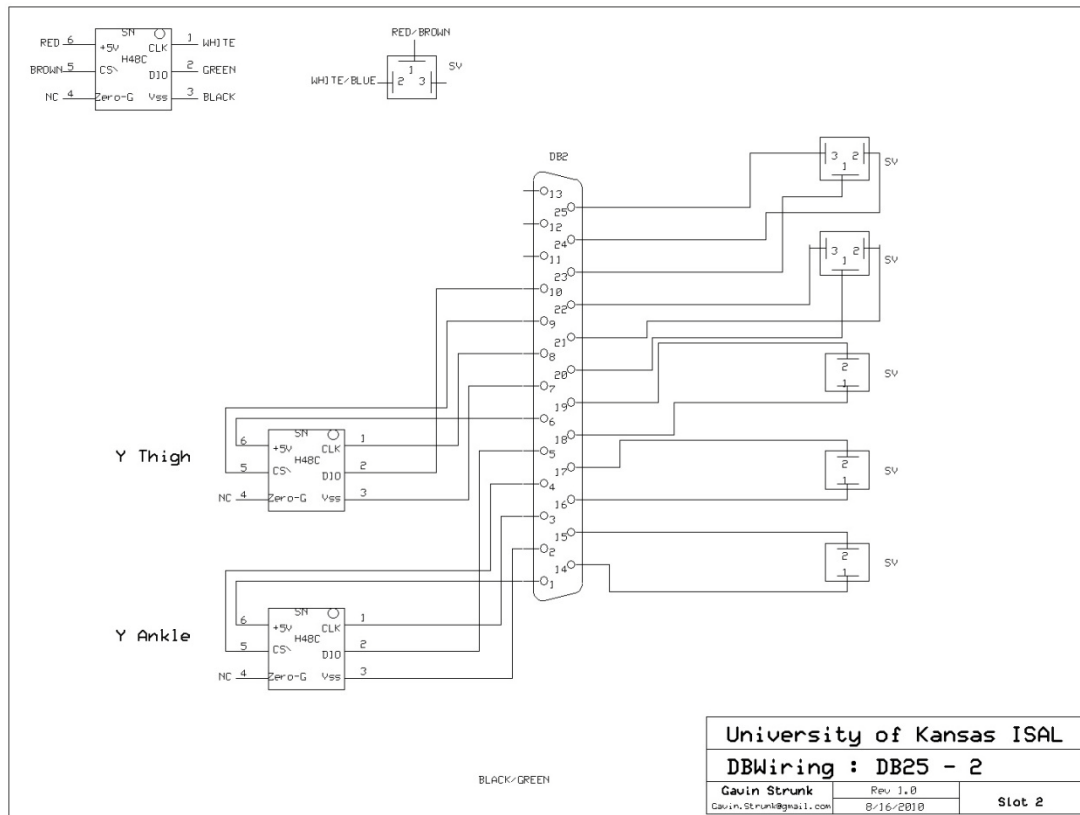


Figure B.7.2: DB25 connector in slot 2.

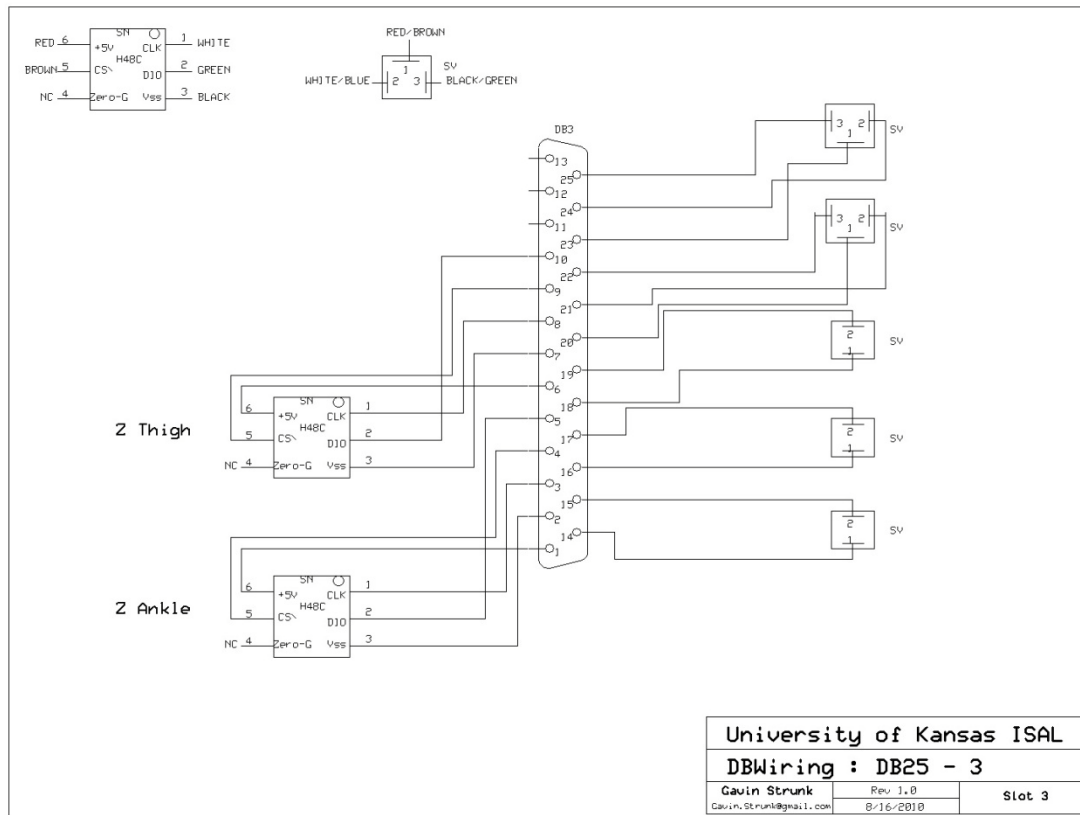


Figure B.7.3: DB25 connector in slot 3.

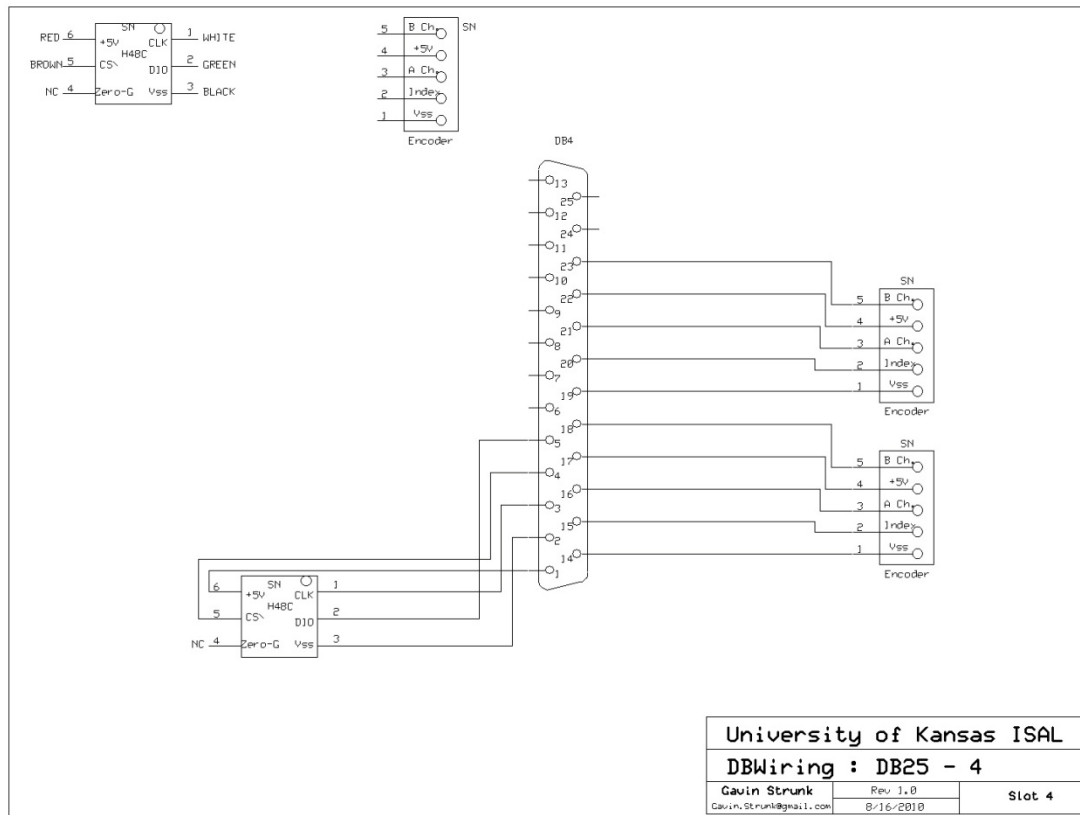


Figure B.7.4: DB25 connector in slot 4.

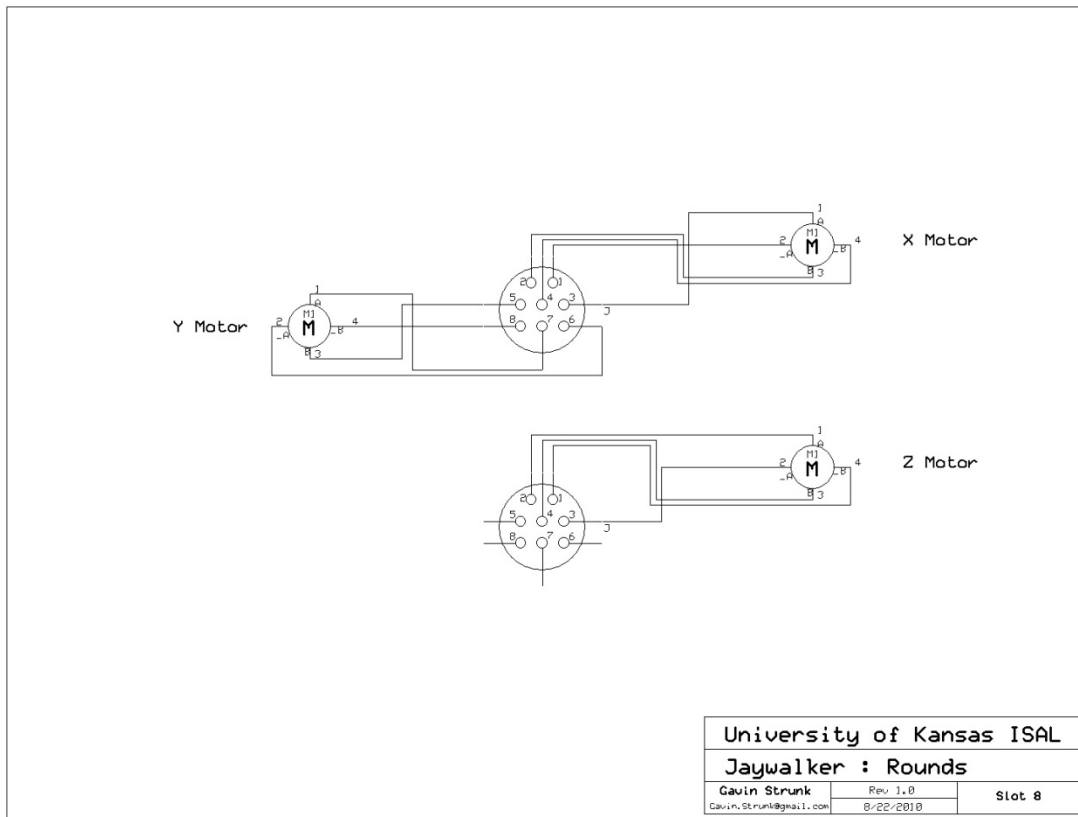


Figure B.7.5: Round connectors in slots 8 and 9.

Appendix C: Jaywalker PCB Layouts

Appendix C contains the circuit board designs for the Jaywalker control system. The board dimensions are not to scale.

C.1: Main Controller Layout

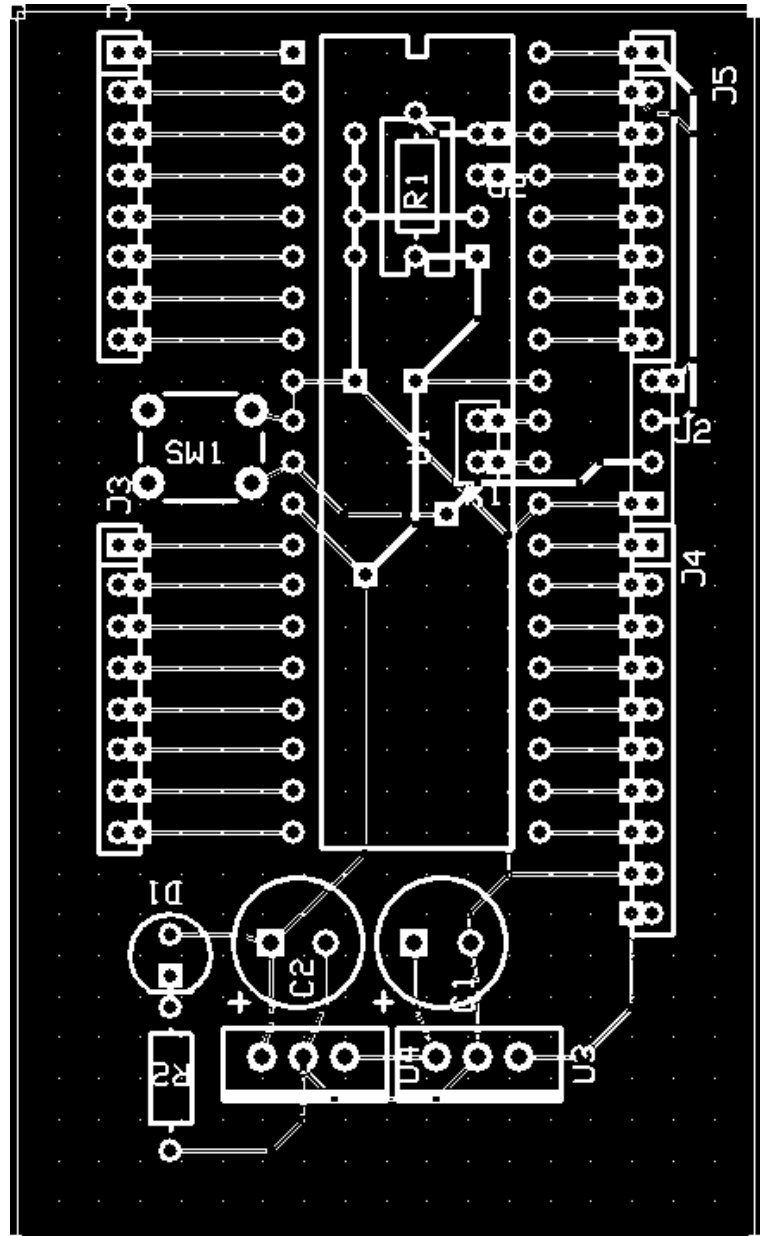


Figure C.1: Main Controller PCB layout in ExpressPCB.

C.2: GUI Controller Layout

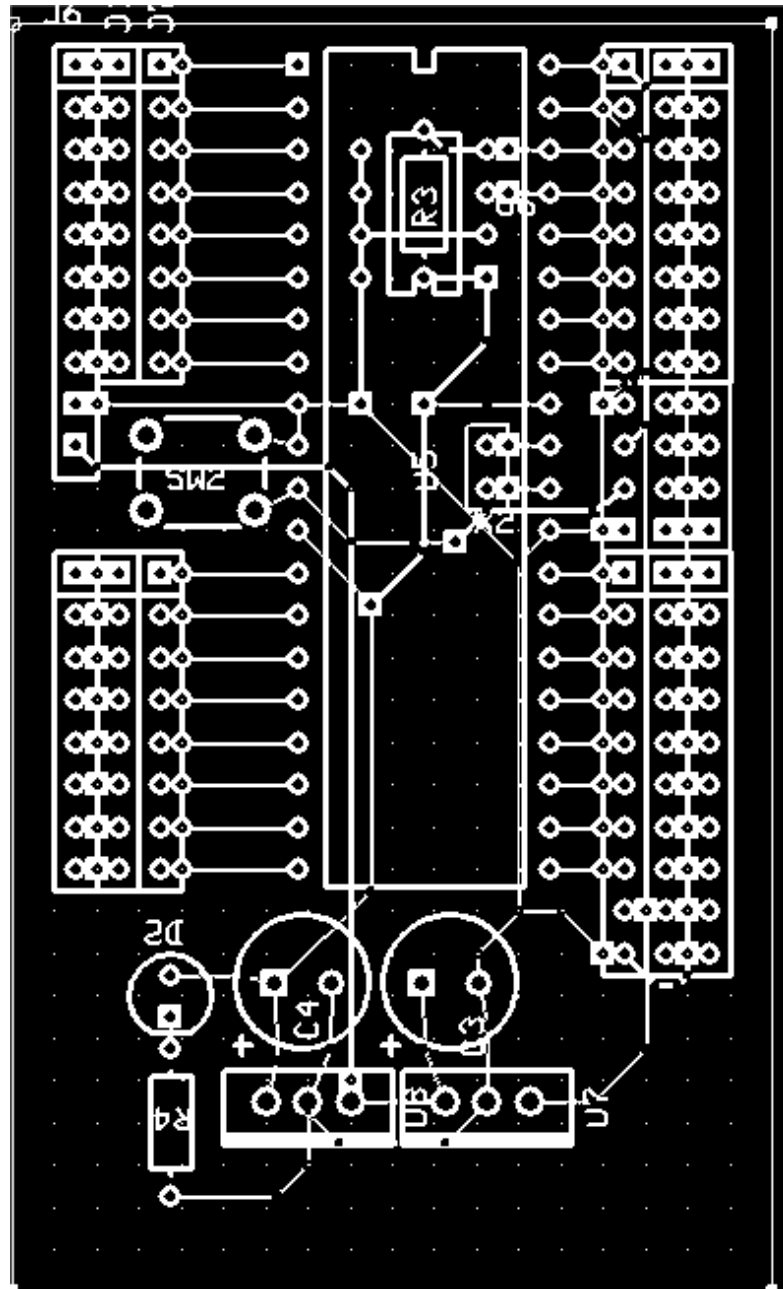


Figure C.2: GUI Controller PCB layout in ExpressPCB.

C.3: Power Controller Layout

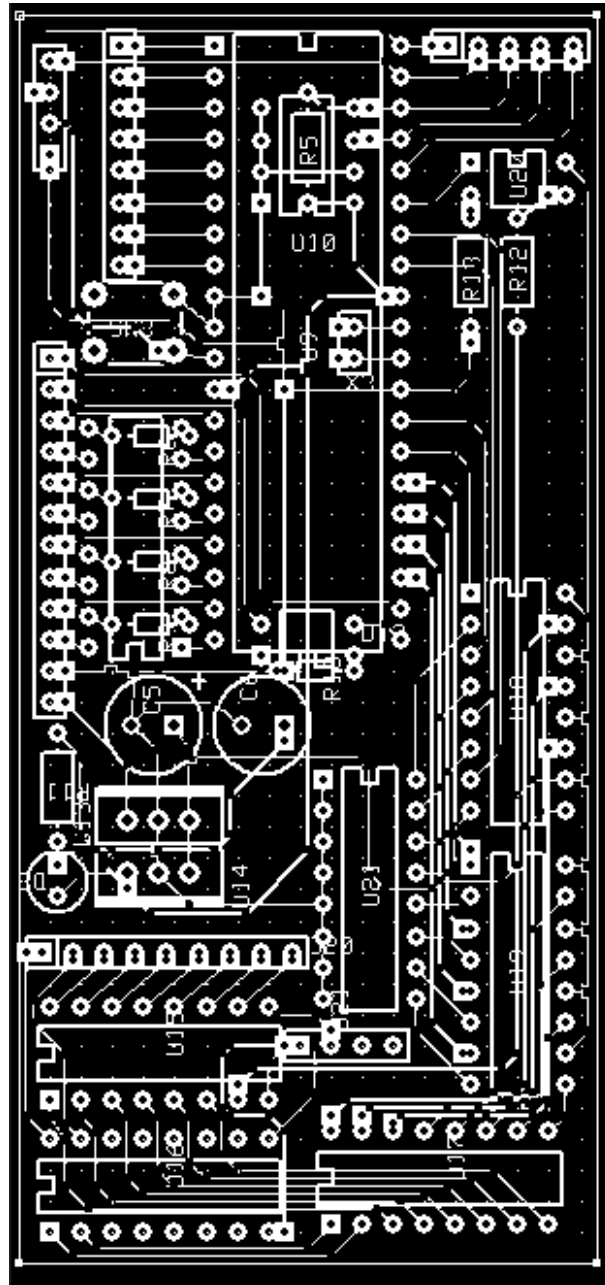


Figure C.3: Power Controller PCB layout in ExpressPCB.

C.4: Foot Controller Layout

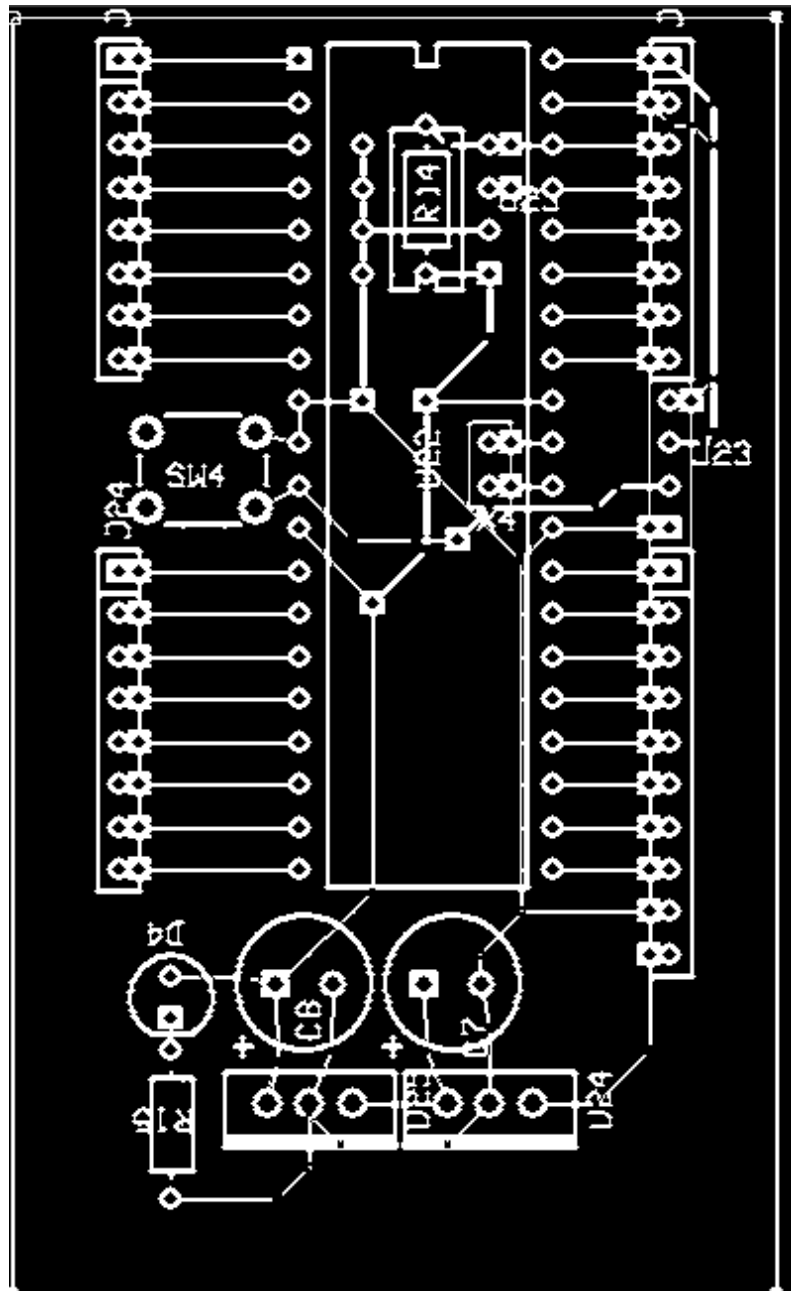


Figure C.4: Foot Controller PCB layout in ExpressPCB.

C.5: Sensor Controller Layout

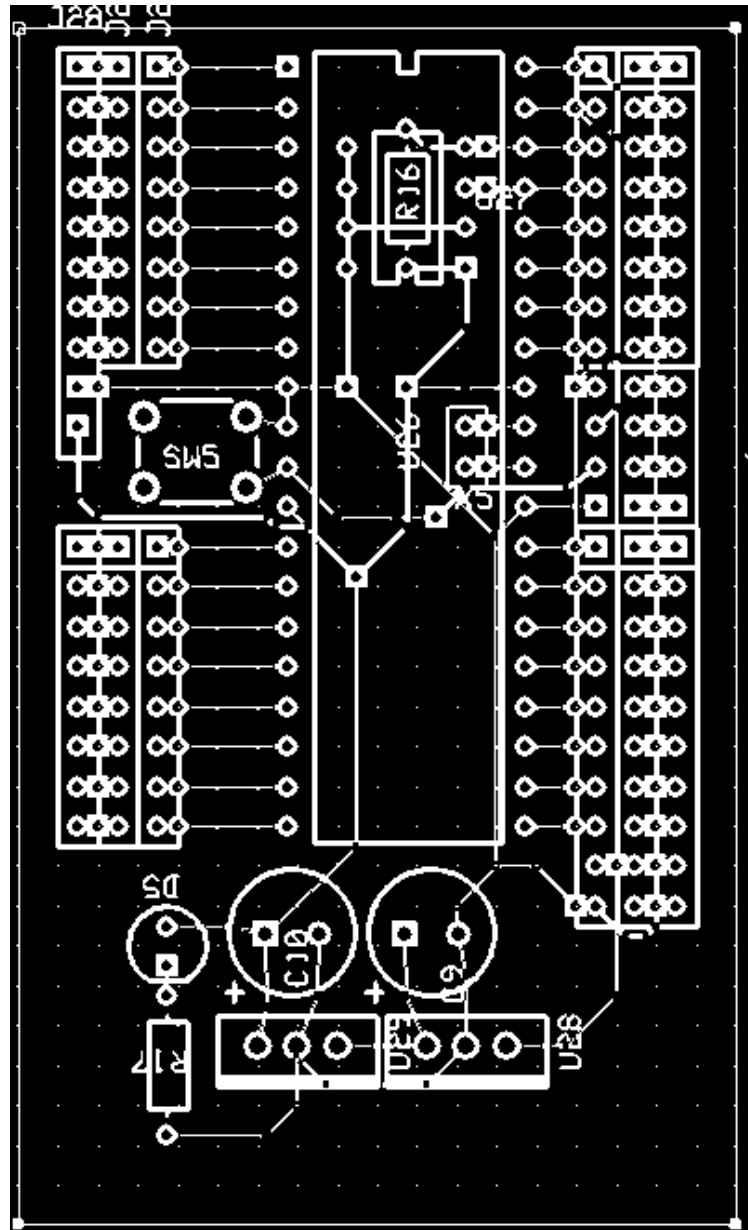


Figure C.5: Sensor Controller PCB layout in ExpressPCB.

C.6: Connector Board Layout

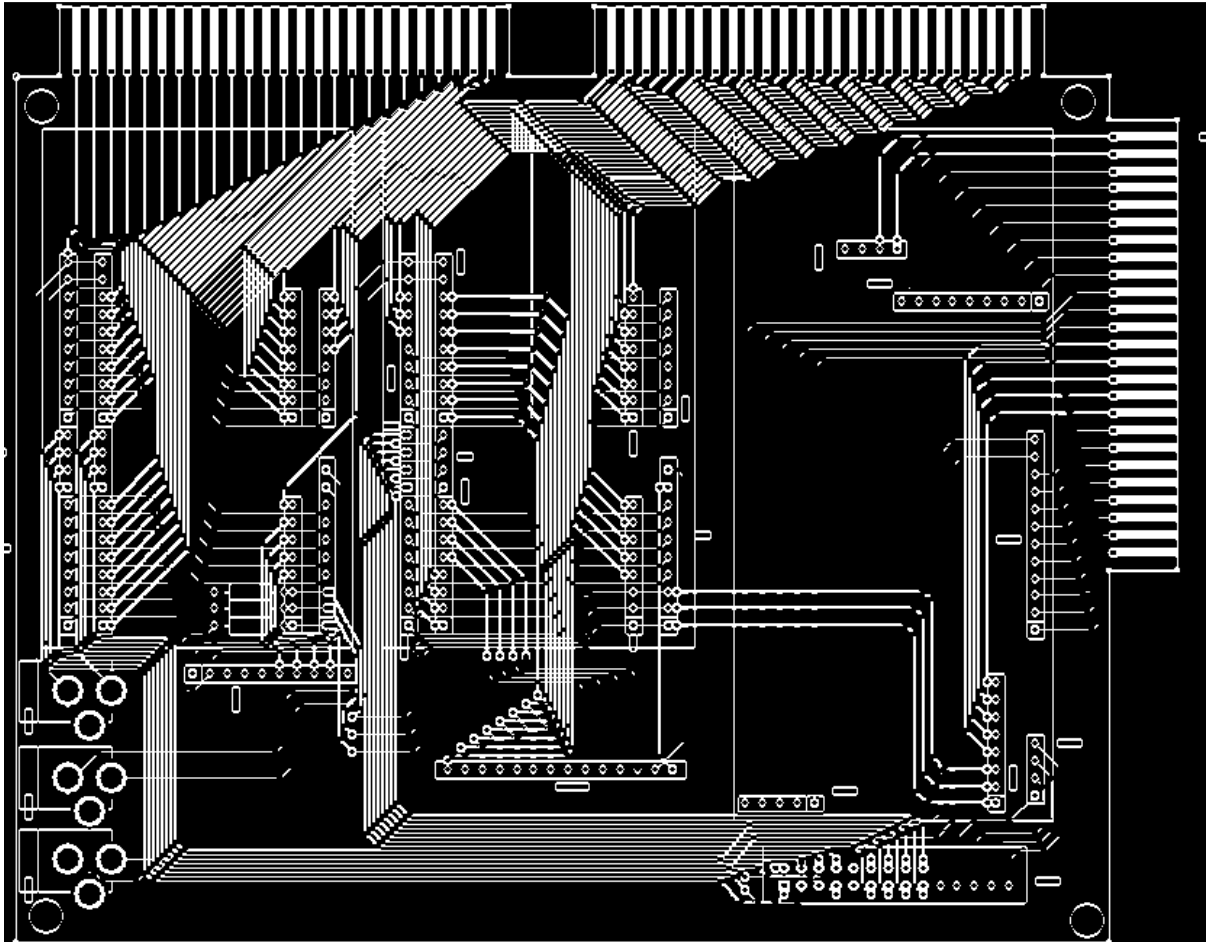


Figure C.6: Connecting Board PCB layout in ExpressPCB.

Appendix D: Jaywalker Program Code

Appendix D contains the code used to interface with the electronic hardware and the logic to take an open loop step.

D.1 MD1 PWM Driver

```
{{
GS_PWM_V1.2.spin
=====
Author : Gavin Strunk
Date   : 17 June 2010
Email  : Gavin.Strunk@gmail.com
University of Kansas ISA Lab
=====
```

Description: This is a finite count PWM driver. It creates a PWM signal of specified frequency on a specified pin for a specified number of cycles.

Schematic: N/A

Notes:

- _pstep = the PWM output pin
- _count = the number of cycles to execute
- _freq = the frequency of the PWM signal
- _duty = sets the duty cycle 0-100 %

Revision History:

- 6/17/10 V1.2 : Add ability to set the duty cycle
- 6/17/10 V1.1 : converted to generic object that has user defined parameters
- 6/17/10 V1.0 : made PWM signal with set parameters

```
}}
CON
  _clkmode = xtal1 + pll16x
  _xinfreq = 5_000_000
VAR
  long pstep,count,freqhi,freqlow,pause
PUB Start(_pstep,_count,_freq,_duty)
  pstep := _pstep
  count := _count
  pause := clkfreq/_freq
  freqhi := _duty*pause/100 'convert specified frequency to corresponding pause
  freqlow := (100-_duty)*pause/100
```

```

cognew (@Toggle, @pstep)
DAT
        org    0
Toggle
        mov    t,par
        rdlong t1,t        'read in the PWM pin
        add    Pin,#1
        shl    Pin, t1

        add    t, #4        'read in the number of cycles to run
        rdlong cycles,t

        add    t, #4        'read in the correct pause for the desired
        rdlong waithi,t        'frequency

        add    t, #4
        rdlong waitlow,t

        cogid  cog        'get the cog id to end program later

        mov    dira, Pin    'set pin to output
        mov    time, cnt    'put value of cnt into time
        add    time, #9    'Add 9 to time to account for the commands
                        'between now and during the loop
:loop

        waitcnt time, waithi    'pause freq
        xor    outa, Pin    'toggle pin state
        waitcnt time, waitlow
        xor    outa, Pin
        djnz   cycles, #:loop    'loop number of cycles

        cogstop cog        'stop cog when finished
=====
=====
Pin      long    0
waithi   long    0
waitlow  long    0
cycles   long    0
time     res     1
t        res     1
t1       res     1
cog      res     1

```

D.2 Jaywalker GUI

```
{{
GS_GUI_Controller_V1.4.spin
=====
Author : Gavin Strunk
Date  : 17 June 2010
Email : Gavin.Strunk@gmail.com
University of Kansas ISA Lab
=====
```

Description: The GUI Controller program is the interface for the Jaywalker and gives control over a preprogrammed walk, as well as a manual mode to test individual systems. It also includes a system check mode for hardware debugging.

Schematic: N/A

Notes: N/A

Revision History:

9/13/10 V1.4: Add direction selection for stepper motors on manual menu
8/24/10 V1.3: Integrated motor operation into a callback between the f and c of U motor
and the debug menu start button with value displays in the H48C window
8/23/10 V1.2: Fixed radio button groupings, start/stop buttons
8/20/10 V1.1: Finished basic outline and bug fixed keyboard refresh, ***bugs found :
manual doesn't save selections, start/stop turn to on/off rather than
switching state
6/17/10 V1.0: Basic outline of the menu structure

```
}}
CON
_clkmode = xtal1 + pll16x
_xinfreq = 5_000_000
```

```
VGA_base   = 16
mouse_dat  = 27
mouse_clk  = 28
keyboard_dat = 29
keyboard_clk = 30
```

```
OBJ
GUI : "GUIBase"
PWM : "GS_PWM_V1.2"
NUM : "Numbers"
```

```
VAR
BYTE vga_rows, vga_cols
BYTE Walk, Systemchk, Manual, Debug
```

```

    BYTE RAD_out,
    RAD_in,RAD_sc1,RAD_sc2,RAD_sc3,RAD_sc4,RAD_sc5,RAD_sc6,RAD_sc7,RAD_all
    BYTE INPF_steps,
    INPF_in,INPF_manuf,INPF_manuc,INPF_manvf,INPF_manvc,INPF_manxf,INPF_manxc,INPF_
    manyf,INPF_manyc,INPF_manzf,INPF_manzc
    BYTE
    TBOX_steps,TBOX_out,TBOX_manmotor,TBOX_manact,TBOX_manenc,TBOX_mansd,TBOX_
    manh48c,TBOX_manft,TBOX_dbggo
    BYTE TBOX_dbgh48c,TBOX_dbgenc,TBOX_dbgls,TBOX_dbgft
    BYTE PUSH_WALKstart,PUSH_SCstart,PUSH_dbggo
    BYTE
    PUSH_mantx,PUSH_manax,PUSH_manay,PUSH_manaz,PUSH_mankx,PUSH_manky,PUSH_m
    ankz,PUSH_manty,PUSH_mantz,PUSH_manencu,PUSH_manencv,PUSH_mansd
    BYTE
    PUSH_manhax,PUSH_manhay,PUSH_manhaz,PUSH_manhcx,PUSH_manhky,PUSH_manhcz,
    PUSH_manhtx,PUSH_manhty,PUSH_manhtz
    BYTE strbuf1[64],buf1[4],buf2[4]
    LONG t1,push,go, freq[5],count[5]

```

PUB Start | cb

push := 0

go := 0

NUM.Init

UIsetup

repeat

cb := GUI.ProcessUI

case cb

Walk: Walk_action

Systemchk: Systemchk_action

Manual: Manual_action

Debug : Debug_action

PUSH_WALKstart : PUSH_go_action(cb,\$0001)

PUSH_SCstart : PUSH_go_action(cb,\$0002)

PUSH_dbggo : PUSH_go_action(cb,\$0004)

PUSH_manax : PUSH_man_action(cb,\$0001)

PUSH_manay : PUSH_man_action(cb,\$0002)

PUSH_manaz : PUSH_man_action(cb,\$0004)

PUSH_mankx : PUSH_man_action(cb,\$0008)

PUSH_manky : PUSH_man_action(cb,\$0010)

PUSH_mankz : PUSH_man_action(cb,\$0020)

PUSH_mantx : PUSH_man_action(cb,\$0040)

PUSH_manty : PUSH_man_action(cb,\$0080)

PUSH_mantz : PUSH_man_action(cb,\$0100)

PUSH_mansd : PUSH_man_action(cb,\$0200)

PUSH_manencu : PUSH_man_action(cb,\$0400)

```

PUSH_manencv : PUSH_man_action(cb,$0800)
PUSH_manhax : PUSH_man_action(cb,$1000)
PUSH_manhay : PUSH_man_action(cb,$2000)
PUSH_manhaz : PUSH_man_action(cb,$4000)
PUSH_manhkx : PUSH_man_action(cb,$0000_8000)
PUSH_manhky : PUSH_man_action(cb,$0001_0000)
PUSH_manhkhz : PUSH_man_action(cb,$0002_0000)
PUSH_manhtx : PUSH_man_action(cb,$0004_0000)
PUSH_manhty : PUSH_man_action(cb,$0008_0000)
PUSH_manhtz : PUSH_man_action(cb,$0010_0000)

```

```

INPF_in : INPF_steps_action(INPF_in)
INPF_manuf : INPF_SetFREQ(INPF_manuf,1)
INPF_manuc : INPF_SetCnt(INPF_manuc,1)

```

PRI UIsetup | tmp

'Start GUIBase object

```
tmp := GUI.Init(VGA_base,mouse_dat,mouse_clk,keyboard_dat,keyboard_clk)
```

'Remember rows and columns for future use

```
vga_rows := ( tmp & $0000FF00 ) >> 8
```

```
vga_cols := tmp & $000000FF
```

'Set screen colors

```
GUI.ClearScreen( %%333, %%333 )      'green on black each is %%RGB 4 levels per R-G-
```

B

repeat tmp from 0 to vga_rows

```
GUI.SetLineColor( tmp, %%333, %%001 )      'Menu Area colour Line 1
```

'Make tabbed menu options

```
GUI.SBOXInit( 0, 0, vga_cols, 3, 0 )      'menu group box
```

```
Walk := GUI.MENUInit(1, 5,string(" WALK "))
```

```
Systemchk := GUI.MENUInit(1,25,string(" System Check "))
```

```
Manual := GUI.MENUInit(1,55,string(" Manual Mode "))
```

```
Debug := GUI.MENUInit(1,80,string(" Debug "))
```

```
GUI.MENUSetStatus(Walk,0)
```

PRI Walk_action | tmp

UIsetup

'Setup walk menu as default view

```

'=====
'Radio buttons to select which leg steps first
GUI.SBOXInit( 10, 4, 18, 7, string("First Step")) 'radio button group box
RAD_out := GUI.RADBInit( 13, 6, 11, string( "Outside Leg" ), 0 )
RAD_in := GUI.RADBInit( 15, 6, 11, string( "Inside Leg " ), 0 )
GUI.RADBSelect( RAD_out, 1 ) 'select outside leg as default

'User input box to define the number of steps with default value 1
INPF_steps := GUI.INPFInit(20,4,20,0,string("Number of Steps"))
GUI.INPFSelect(INPF_steps,0)

'Start/stop button
PUSH_WALKstart := GUI.PUSHInit(25,4,string(" Start "))

PRI Systemchk_action | tmp
'refresh display
UIsetup

'setup output box
TBOX_out := GUI.TBOXInit(8,8,40,20,1,string("Output"))
'GUI.TBOXPrint(TBOX_out, ,0)
'setup input box for commands
INPF_in := GUI.INPFInit(27,8,40,1,string("Input"))

'setup radio buttons to toggle slot selection
GUI.SBOXInit(32,8,17,12,string("Slot Selection"))
RAD_sc1 := GUI.RADBInit(35,10,7,string("Slot 1"),1)
RAD_sc2 := GUI.RADBInit(36,10,7,string("Slot 2"),1)
RAD_sc3 := GUI.RADBInit(37,10,7,string("Slot 3"),1)
RAD_sc4 := GUI.RADBInit(38,10,7,string("Slot 4"),1)
RAD_sc5 := GUI.RADBInit(39,10,7,string("Slot 5"),1)
RAD_sc6 := GUI.RADBInit(40,10,7,string("Slot 6"),1)
RAD_sc7 := GUI.RADBInit(41,10,7,string("Slot 7"),1)
RAD_all := GUI.RADBInit(42,10,11,string("All Systems"),1)
GUI.RADBSelect(RAD_all,1)

'setup start/stop buttons
PUSH_SCstart := GUI.PUSHInit(32,35,string(" Start "))

PRI Manual_action
'refresh display
UIsetup

'Motors Submenu
TBOX_manmotor := GUI.TBOXInit(6,6,30,15,1,string(" Motors "))
GUI.TBOXPrint(TBOX_manmotor,string(" "),1)
GUI.TBOXPrint(TBOX_manmotor,string("U Motor "),12)

```

```

GUI.TBOXPrint(TBOX_manmotor,string(" "),1)
GUI.TBOXPrint(TBOX_manmotor,string("V Motor  "),12)
GUI.TBOXPrint(TBOX_manmotor,string(" "),1)
GUI.TBOXPrint(TBOX_manmotor,string("X Motor  "),12)
GUI.TBOXPrint(TBOX_manmotor,string(" "),1)
GUI.TBOXPrint(TBOX_manmotor,string("Y Motor  "),12)
GUI.TBOXPrint(TBOX_manmotor,string(" "),1)
GUI.TBOXPrint(TBOX_manmotor,string("Z Motor  "),12)

'Motor inputs
INPF_manuf := GUI.INPFInit(9,15,9,0,string("F"))
INPF_manuc := GUI.INPFInit(9,25,10,0,string("C"))
INPF_manvf := GUI.INPFInit(11,15,9,1,string("F"))
INPF_manvc := GUI.INPFInit(11,25,10,1,string("C"))
INPF_manxf := GUI.INPFInit(13,15,9,1,string("F"))
INPF_manxc := GUI.INPFInit(13,25,10,1,string("C"))
INPF_manyf := GUI.INPFInit(15,15,9,1,string("F"))
INPF_manyc := GUI.INPFInit(15,25,10,1,string("C"))
INPF_manzf := GUI.INPFInit(17,15,9,1,string("F"))
INPF_manzc := GUI.INPFInit(17,25,10,1,string("C"))

'Actuator Submenu
TBOX_manact := GUI.TBOXInit(6,50,30,14,1,string("    Actuators  "))
GUI.TBOXPrint(TBOX_manact,string("    X Y Z"),26)
GUI.TBOXPrint(TBOX_manact,string(" "),1)
GUI.TBOXPrint(TBOX_manact,string("Ankle  "),7)
GUI.TBOXPrint(TBOX_manact,string(" "),1)
GUI.TBOXPrint(TBOX_manact,string(" "),1)
GUI.TBOXPrint(TBOX_manact,string("Knee  "),7)
repeat 2
    GUI.TBOXPrint(TBOX_manact,string(" "),1)
GUI.TBOXPrint(TBOX_manact,string("Thigh  "),7)

'Actuator inputs
PUSH_manax := GUI.PUSHInit(10,63,string("Off"))
PUSH_manay := GUI.PUSHInit(10,68,string("Off"))
PUSH_manaz := GUI.PUSHInit(10,73,string("Off"))
PUSH_mankx := GUI.PUSHInit(13,63,string("Off"))
PUSH_manky := GUI.PUSHInit(13,68,string("Off"))
PUSH_mankz := GUI.PUSHInit(13,73,string("Off"))
PUSH_mantx := GUI.PUSHInit(16,63,string("Off"))
PUSH_manty := GUI.PUSHInit(16,68,string("Off"))
PUSH_mantz := GUI.PUSHInit(16,73,string("Off"))

'Encoder Submenu
TBOX_manenc := GUI.TBOXInit(25,6,15,10,1,string(" Encoders "))
GUI.TBOXPrint(TBOX_manenc,string(" "),1)

```



```

GUI.TBOXPrint(TBOX_manenc,string("U Motor"),8)
repeat 2
  GUI.TBOXPrint(TBOX_manenc,string(" "),1)
GUI.TBOXPrint(TBOX_manenc,string("V Motor"),8)

'Encoder inputs
PUSH_manencu := GUI.PUSHInit(28,15,string("Off"))
PUSH_manencv := GUI.PUSHInit(31,15,string("Off"))

'SD On/Off
TBOX_mansd := GUI.TBOXInit(25,27,9,7,1,string("SD Card"))
PUSH_mansd := GUI.PUSHInit(28,29,string("Off"))

'H48C Submenu
TBOX_manh48c := GUI.TBOXInit(25,50,30,14,1,string("  H48C Accelerometers"))
GUI.TBOXPrint(TBOX_manh48c,string("      X  Y  Z"),26)
GUI.TBOXPrint(TBOX_manh48c,string(" "),1)
GUI.TBOXPrint(TBOX_manh48c,string("Ankle  "),7)
GUI.TBOXPrint(TBOX_manh48c,string(" "),1)
GUI.TBOXPrint(TBOX_manh48c,string(" "),1)
GUI.TBOXPrint(TBOX_manh48c,string("Knee  "),7)
repeat 2
  GUI.TBOXPrint(TBOX_manh48c,string(" "),1)
GUI.TBOXPrint(TBOX_manh48c,string("Thigh  "),7)

'H48c inputs
PUSH_manhax := GUI.PUSHInit(29,63,string("Off"))
PUSH_manhay := GUI.PUSHInit(29,68,string("Off"))
PUSH_manhaz := GUI.PUSHInit(29,73,string("Off"))
PUSH_manhkx := GUI.PUSHInit(32,63,string("Off"))
PUSH_manhky := GUI.PUSHInit(32,68,string("Off"))
PUSH_manhkz := GUI.PUSHInit(32,73,string("Off"))
PUSH_manhtx := GUI.PUSHInit(35,63,string("Off"))
PUSH_manhty := GUI.PUSHInit(35,68,string("Off"))
PUSH_manhtz := GUI.PUSHInit(35,73,string("Off"))

'foot Submenu
TBOX_manft := GUI.TBOXInit(36,6,30,10,1,string("  Foot Control  "))
GUI.TBOXPrint(TBOX_manft,string("For future dev of foot"),23)

PRI Debug_action
'refresh display
UIsetup

'start/stop button
TBOX_dbggo := GUI.TBOXInit(6,6,20,6,0,0)

```

```

GUI.TBOXPrint(TBOX_dbggo,string("  Manual  "),13)
PUSH_dbggo := GUI.PUSHInit(8,10,string(" Start "))

'H48C readout
TBOX_dbgh48c := GUI.TBOXInit(6,50,40,20,1,string("H48C Accelerometers"))

'Encoder readout
TBOX_dbgenc := GUI.TBOXInit(29,50,40,7,1,string("Quadrature Encoders"))

'Limit switch readout
TBOX_dbgls := GUI.TBOXInit(37,50,40,7,1,string("Limit Switches "))

'Foot sensor readout
TBOX_dbgft := GUI.TBOXInit(15,6,30,20,1,string("Foot Sensors"))

PRI INPF_steps_action(guid)
  GUI.INPFGetString(guid, @strbuf1)
  GUI.TBOXPrint(TBOX_out, @strbuf1,0)

PRI PUSH_man_action(guid,mask)| temp
  temp := push
  push ^= mask

  if temp < push
    GUI.PUSHSetText(guid,string("On "))
  else
    GUI.PUSHSetText(guid,string("Off"))

PRI PUSH_go_action(guid,bmask) | temp
  temp := go
  go ^= bmask

  if temp < go
    GUI.PUSHSetText(guid,string(" Stop "))
    GUI.TBOXPrint(TBOX_dbgh48c,@buf1,0)
    GUI.TBOXPrint(TBOX_dbgh48c,@buf2,0)
    dira[4]~~
    outa[4]~
    PWM.Start(3,count[0],freq[0],50)
  else
    GUI.PUSHSetText(guid,string(" Start "))

PRI INPF_SetFreq(guid,motor)| t,f
  GUI.INPFGetString(guid,@buf1)
  t := NUM.FromStr(@buf1,10)
  freq[0] := t

```

```
PRI INPF_SetCnt(guid,motor)|t,c  
  GUI.INPFGetString(guid,@buf2)  
  t := NUM.FromStr(@buf2,10)  
  count[0] := t
```

D.3 MMPS I²C Driver

```
{{ GS_MMPSI2C_V1.2.spin
```

```
=====
```

```
Author : Gavin Strunk
```

```
Date  : 16 October 2010
```

```
Email : Gavin.Strunk@gmail.com
```

```
University of Kansas ISA Lab
```

```
=====
```

Description: This is a multi-master parallel slave (MMPS) I2C communication protocol.

It has self contained functionality for the master and the slaves. For our purposes the master will only write due to the nature of the application, but functionality will be included to read with the master for robustness.

Notes:

- 1.) High Priority ACK (HPA) is implemented in this program, could also implement ACK-In-Turn (AIT) or ACK-As-Available (AAA) and handle arbitration. Speed is and issue so the HPA strategy was implemented to reduce ACK time.

Revision History

10/26/2010 V1.2 : Now incorporate ACK and address and data sending and receiving

10/26/2010 V1.1 : Basic setup complete, start, stop, write read

10/16/2010 V1.0 : write startup functions and basic read and write

```
}}
```

```
CON
```

```
_clkmode = xtal1 + pll16x
```

```
_xinfreq = 5_000_000
```

```
"Pin descriptions
```

```
SDA = 0
```

```
SCL = 1
```

```
"I2C constants
```

```
ACK = 1
```

```
NACK = 0
```

```
"Pin state declarations
```

```
High = 1
```

```
Low  = 0
```

```
OBJ
```

```
serial: "FullDuplexSerial"
```

```
VAR
```

```
LONG a,d,b
BYTE temp
```

```
PUB Init|go
```

```
go := 1
```

```
serial.start(31,30,0,57600)
waitcnt(5*clkfreq + cnt)
```

```
if go == 1
  Master
  repeat
```

```
  Slave
  repeat
```

```
PUB Master|ac
```

'The function identifies the master and sends the start sequence

'Steps:

```
' 1. Send Start bit    *
' 2. Send command      *
' 3. Wait for ACK
' 4. Send/Receive Byte
' 5. Send ACK
' 6. Stop bit          *
```

```
a := ina
serial.bin(a,32)
serial.str(string($0d))
```

```
Start
```

```
ac := Write_Command($08)
```

```
if ac == ACK
  Write_Byte($1e)
```

```
Stop
```

```
a := ina
serial.bin(a,32)
```

```
PUB Slave |addr,t1
```

'This function identifies the slave and immediately begins listening

'Steps:

```
' 1. Listen for start bit *
```

```
' 2. Listen for command  *
' 3. ACK or ignore
' 4. Receive/Send Byte
' 5. Receive ACK
' 6. Listen for stop bit
serial.str(string("started"))
repeat until b == 3
  b := ina & $03
```

```
serial.str(string("lines high"))
```

Listening

```
addr := Read_Command
```

```
t1 := NACK
if addr == $08
  t1 := ACK
```

```
Send_Ack(t1)
Read_Byte
```

```
serial.dec(temp)
serial.str(string("this is working"))
```

PRI Start

```
'Private function call by the master to signal the start of a transfer
dira[SCL]~
dira[SDA]~~
outa[SDA]~
```

```
return
```

PRI Stop

```
'Private function called by the master to signal the end of a transfer
```

```
dira[SCL]~
dira[SDA]~~
outa[SDA]~~
return
```

PRI Write_Byte(data)

```
'Private function called by master to send a byte
dira[SCL]~~
outa[SCL]~
serial.str(string($0d))
```

```

a := ina
serial.bin(a,32)
serial.str(string($0d))
data <<= 24                                'shift data left 24 times
repeat 8
  outa[SDA] := (data <=1) & 1                'rotate data left 1 MSB first
  outa[SCL]~~
  waitcnt(clkfreq/100 + cnt)                'clock pulse SCL
  outa[SCL]~

return
PRI Read_Byte
'Private function called by master or slave to read a byte
serial.str(string("reading"))
temp :=0
repeat until ina[SCL] == low

repeat 8
  repeat until ina[SCL] == high
  temp := (temp << 1) | ina[SDA]
  repeat until ina[SCL] == low

return
PRI Listening|c
'Private function to put slave in listening mode
'repeat until ina[SDA] == high & ina[SCL]==high
serial.str(string("listening"))
repeat until c == 2
  c := ina & $03

return

PRI Write_Command(data)|t,tt
'Private function called by master to write command. Implemented seperate from
Write_Byte
'because Write_Command checks for arbitration.
  Write_Byte(data)

'receive ack bit
  dira[SDA]~
  repeat until tt == $02
  tt := ina & $02

t := ina[SCL]

```

```
return t
```

```
PRI Read_Command
```

```
Read_Byte
```

```
return
```

```
PRI Send_Ack(state)
```

```
dira[SDA]~~
```

```
dira[SCL]~~
```

```
outa[SDA] := state
```

```
outa[SCL]~~
```

```
waitcnt(clkfreq/100 + cnt)
```

```
outa[SCL]~
```

```
return
```


Appendix E: Jaywalker GUI Screenshots

Appendix E show screenshots of the GUI menu produced by the code presented in Appendix D.

E.1 Walk Menu

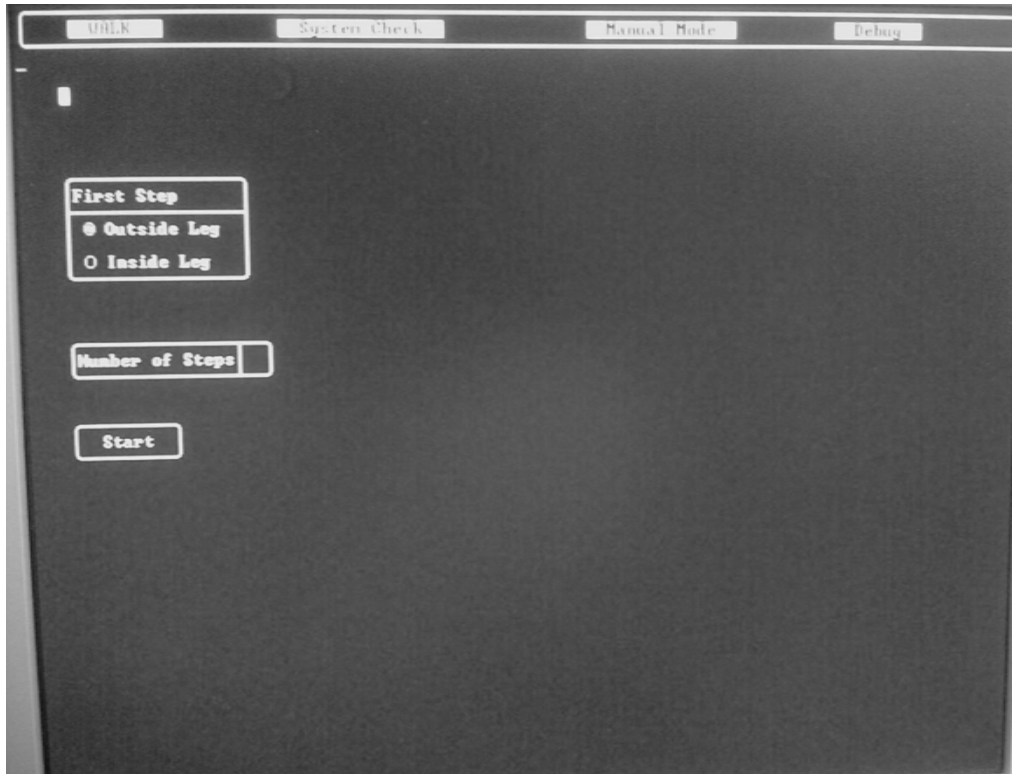


Figure E.1: Screenshot of the Walk Menu in the Jaywalker GUI.

E.2 System Check Menu

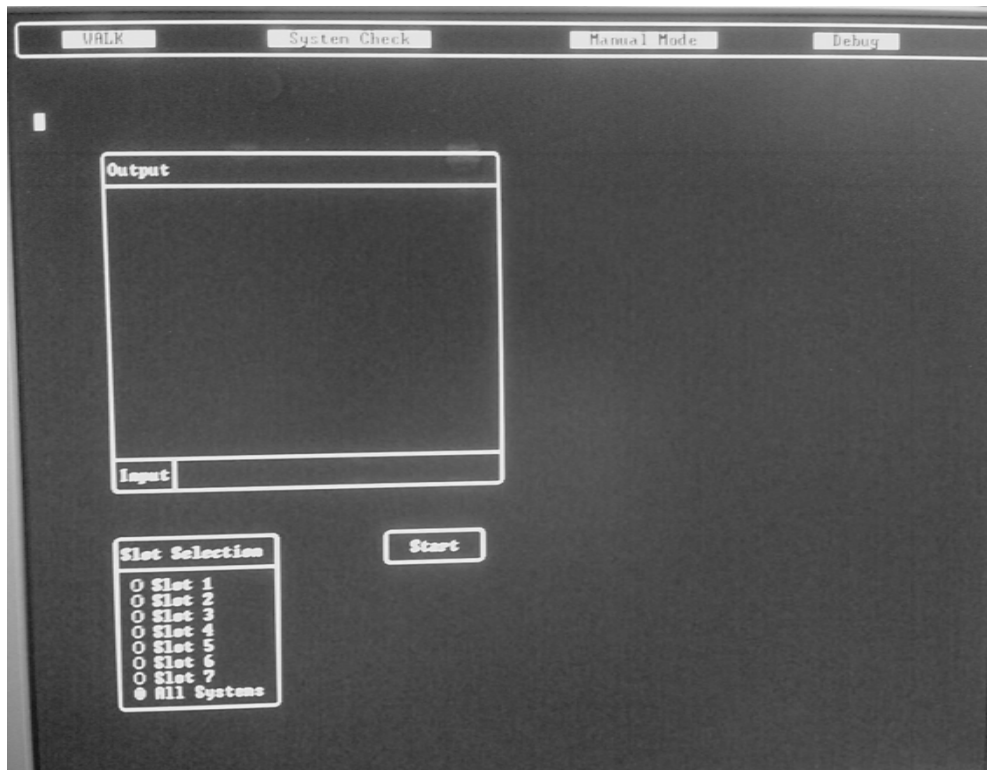


Figure E.2: Screenshot of the System Check Menu in the Jaywalker GUI.

E.3 Manual Mode Menu

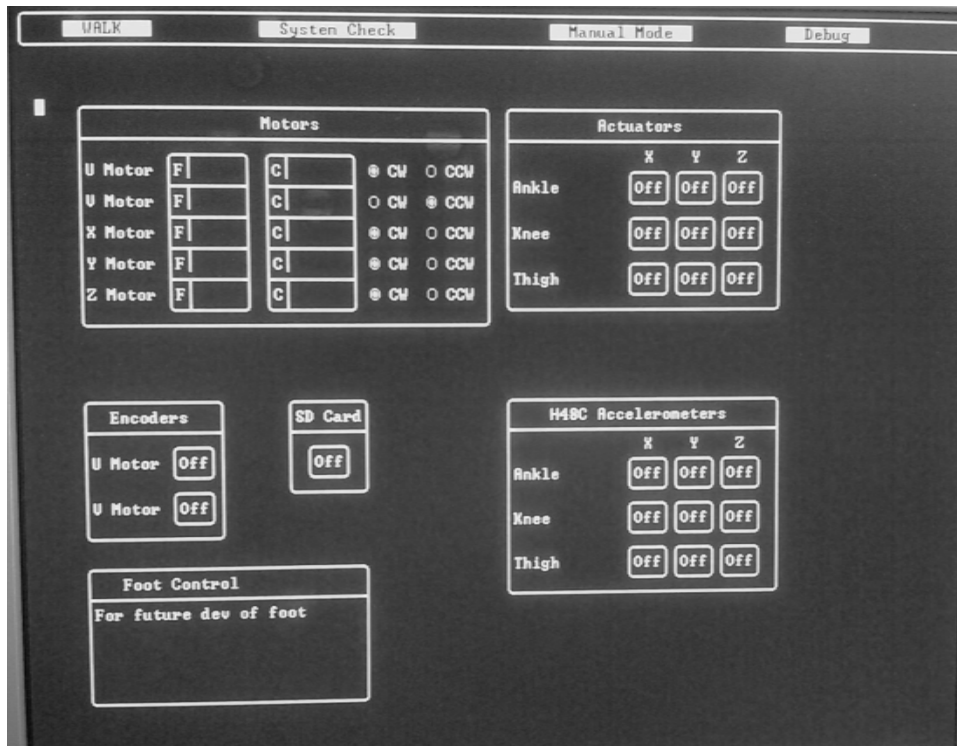


Figure E.3: Screenshot of the Manual Mode Menu in the Jaywalker GUI.

E.4 Debug Menu

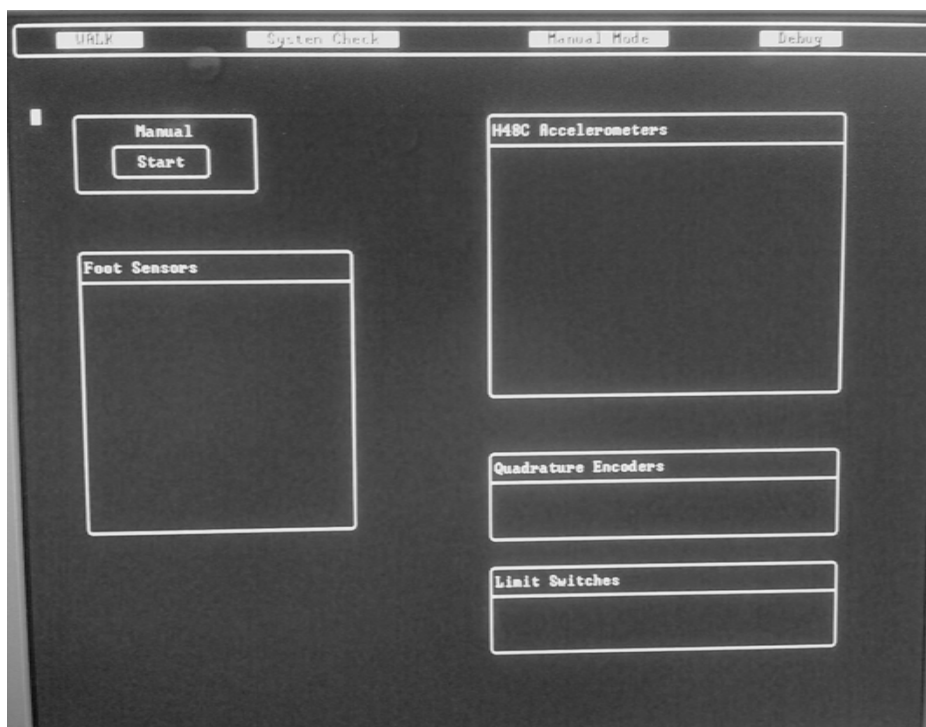


Figure E.4: Screenshot of the Debug Menu in the Jaywalker GUI.

Appendix F: Jaywalker Step Screenshots

The still frames in Appendix F are every third frame from walking video. The images show the middle leg step from toe-off to heel strike.

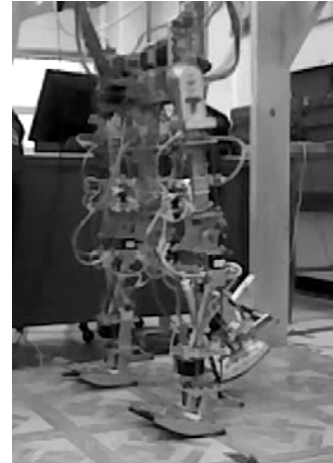
F.1: Middle Leg Step Screenshots



1



3



5



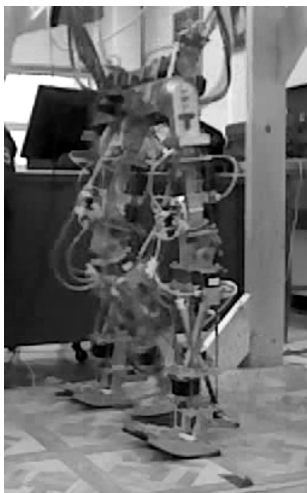
2



4



6



7



9



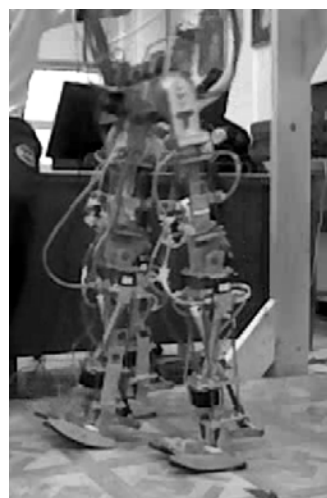
11



8



10



12



13